Space Project Mission Operations Control Architecture (SuperMOCA)

SuperMOCA SYSTEM CONCEPT

Annex 1

Control Interface Specification

April 1996

ORIENTATION

The goal of the **S**pace **Pr**oject **M**ission **O**perations **C**ontrol **A**rchitecture ("SuperMOCA") is to create a set of implementation-independent open specifications for the standardized monitor and control of space mission systems. Monitoring is the observation of the performance of the activities of these systems. Controlling is the direction of the activities performed by these systems. Overall, monitor and control is the function that orchestrates the activities of the components of each of the systems so as to make the mission work. Space mission systems include:

spacecraft and launch vehicles that are in flight, and;
their supporting ground infrastructure, including launch pad facilities and ground terminals used for tracking and data acquisition.

The SuperMOCA system concept documents consist of the following:

SuperMOCA System Concept, Volume 1: Rationale and Overview
SuperMOCA System Concept, Volume 2: Architecture
SuperMOCA System Concept, Volume 3: Operations Concepts
SuperMOCA System Concept, Annex 1: Control Interface Specification
SuperMOCA System Concept, Annex 2: Space Messaging Service (SMS) Service Specification
SuperMOCA System Concept, Annex 3: Communications Architecture
SuperMOCA System Concept, Ancillary Document 1: Ground Terminal Reference Model
SuperMOCA System Concept, Ancillary Document 2: Operations Center to Ground Terminal Scenarios
SuperMOCA System Concept, Ancillary Document 3: Operations Center to Ground Terminal – Comparison of Open Protocols

These documents are maintained by the custodian named below. Comments and questions to the custodian are welcomed.

Michael K. Jones
MS 301-235
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
Voice: 818-354-3918
FAX: 818-354-9068
E-mail: michael.k.jones@jpl.nasa.gov

Contents

1. INTRODUCTION AND OVERVIEW

1.1. INTRODUCTION

The Space Project Mission Operations Control Architecture ("SuperMOCA") project has the goal of developing open international standard specifications for the activities associated with the monitor and control of space vehicles and their supporting ground systems. The SuperMOCA addresses the mechanisms required to conduct a telecommunicated process control dialog between a human (or machine) user and remote, distributed space mission systems.

The standard Application layer language and protocol being advocated by the SuperMOCA currently contains four elements:

- Control Interface — The Control Interface allows the human (or automated) controller to specify and monitor the desired sequence of operations to be conducted in a remote system.
- Space Messaging System — The Space Messaging System (SMS) translates machine-readable command calls from the Control Interface into standard-syntax messages that invoke the desired actions in the remote end system and returns standard-syntax response information to the controller.
- Decision Support Logic — The Decision Support Logic allows rules for command execution to be programmed into a distributed inference engine (which may be located wholly on the ground, wholly in space, or partitioned in varying ways between the two).
- Information Architecture — The Information Architecture provides the mechanism whereby the precise characteristics of a particular concrete controllable device can be captured and described in standardized, abstract terms.

This document addresses the Control Interface. Part B of this document provides a complete specification for a Control Interface Language (CIL), a next generation language within the family of interface languages designed for use in space mission test and operations. Part C provides a specification of the standard objects needed by the CIL. Part A provides a brief introduction to the CIL and its standard objects. Further information on the rationale, requirements, and design of the CIL will be furnished in a separate document.

1.2. CONTROL INTERFACE LANGUAGE

1.2.1. BACKGROUND

Specialized languages are often part of the human-computer interface to ground systems used for testing, launching and operating spacecraft. Examples of such languages include the Systems Test and Operations Language (STOL) used by NASA]; the Ground Operations and Aerospace Language (GOAL) used in space shuttle launch operations; and the European Test and Operations Language (ETOL) used by ESA. These languages provide imperative directives that can be typed at a keyboard, allowing operators to configure ground systems and to issue commands to a spacecraft. Increasingly, however, graphical user interfaces are used to perform these functions. This has lead some to argue that special test and operations languages are no longer needed. But these languages provide an additional capability that is not easily supplanted by graphical interfaces: complex test and operations procedures written in these languages can be prepared well in advance and run by operators at an appropriate time. Experience has shown that these procedures substantially reduce both operator workload and the potential for errors during testing and mission operations.

Because of the importance of procedures, specialized test and operations languages are not likely to disappear. However, existing languages are clearly showing their age: extending first-generation languages like STOL to meet the needs of newer missions is proving difficult. To overcome these difficulties, the SuperMOCA project undertook an effort to provide a specification for a next-generation language for space mission operations. Test and operations languages in current use, as well as numerous programming languages and scripting languages, were examined but this investigation failed to locate a satisfactory existing candidate for the next-generation operations language. Some second-generation test and operations languages, for example the Colorado-enhanced Systems Test and Operations Language (CSTOL) and the Spacecraft Command Language (SCL), are definitely more robust than the earliest generation of operations languages like STOL and GOAL, but these languages are bound rather tightly to unique software implementations which are not conducive to an open system standard. Programming languages and scripting languages tend to be aimed at software professionals and are not particularly suitable for the engineers, operators, scientists, astronauts, etc. who are expected to be the users of a next-generation operations language. It was therefore decided to produce a specification for a new language, currently known simply as the Control Interface Language (CIL). This language is based upon the specification that was developed several years ago for the Space Station Freedom User Interface Language (UIL). A complete UIL specification and several validating parsers were developed, but the UIL was not employed by the space station project. While the CIL specification is rooted in the original UIL, aspects of the language that were specific to the space station have been removed, several new capabilities added, and numerous refinements have been made.

Unique aspects of the CIL include the following:

The language is object oriented. The ground systems and spacecraft that a specific implementation of the CIL will control are described in object encyclopedias provided through the Information Architecture. The CIL uses the information in these databases to determine which actions it can — and cannot — perform. This allows the language to easily expand to handle new applications.

CIL commands have a syntax that is close to natural English. Thus directives can be spoken as well as entered through a keyboard, opening up new applications for operations languages. For example, an astronaut working on a construction task in space could use spoken CIL commands to control the actions of a robot assistant.

The language can be used to control systems at the low-level of individual actuators and sensors — for example, OPEN VALVE 1 — and at a high level suitable for systems that have intelligence to carry out complex actions — for example, REORIENT SPACECRAFT TO NADIR POINTING.

CIL commands can be translated into the Space Messaging System as well as other protocols and transmitted to the local or remote software and hardware of a ground system or spacecraft. The CIL is not tied to any specific underlying software for processing commands. For example, CIL commands to a spacecraft under test might be implemented using the SMS, although CIL commands to control the associated ground support equipment might be implemented in a totally different way.

The language has extensive data manipulation capabilities so that decisions for future action can be based upon an analysis of the current state of the system under control. These capabilities complement the capabilities of the Decision Support Logic. The CIL provides *fully unitized arithmetic*, where each operand in an arithmetic expression includes its unit of measure. The CIL will ensure that an expression makes sense, given the types of measurements involved. For example, an operand expressed in feet can be added to an operand in meters since both are a measure of distance, but an operand representing a distance cannot be added to an operand expressed in degrees Celsius.

1.2.2. THE USES AND USERS OF THE CIL

The CIL is designed as a potential replacement language for the languages currently used in spacecraft test and operations systems. In this setting the CIL will be used by spacecraft test engineers and operations personnel. The language is also suitable for use by the engineers and scientists who design spacecraft subsystems and payloads. It is assumed that many of these individuals are not (and do not want to be) computer programmers, and so a language that uses the concise but often cryptic notation of a programming language is not advantageous or appropriate for these users. Toward that end, the language has been designed to have the flavor of natural English.

The language can be used to control the hardware and software of ground systems needed for space missions. This includes test equipment, launch facilities, and tracking and receiving stations. The CIL is designed for use within even the most complex such systems. This document does not specify or restrict how ground system software, hardware or networks are implemented.

The CIL may be used onboard both unmanned and manned spacecraft. CIL procedures can be loaded into a processor onboard an unmanned spacecraft and executed to carry out complex operations. In this regard, CIL procedures offer an alternative to the traditional mechanism of preparing sequences of binary commands on the ground and then loading them into an onboard command store for later execution. Procedures can also be used on manned spacecraft. Additionally, for manned spacecraft the CIL may be used by astronauts to communicate with complex onboard systems. This communication can be accomplished through keyboard entry or using a voice recognition system. The latter capability allows, for example, an astronaut whose hands are engaged in an experiment, or even encased within a space suit, to query and control automated systems.

While the main motivation for the CIL has been to address the unique needs of spacecraft test and operations, care has been taken to make the language general enough for use in a wide variety of terrestrial process control applications. For example, the language could be used to implement the operations procedures for a power plant or to provide part of the user interface to automated equipment on a factory floor. For this reason, the jargon that is often used by spacecraft engineers (for example, referring to “discrete” and “analog” measurements within a “telemetry stream”) is omitted from the language in favor of more general and widely-understood terms.

1.3. OBJECTS

In a broad sense, the CIL is a language that is designed for two purposes: to make it easy for users to control objects; and to sense the state of objects and to act upon that state. The CIL controls objects by issuing commands to them and it senses the state of objects through expressions. Numerous CIL statements (for example, the While and For statements) use the results of expression to guide the processing of statements within a CIL procedure. The syntax of expressions and commands are fully defined in the language specification, but most of the semantics of commands and expressions are not included in Part B of this document because whatever happens when an object is issued a command or is acted upon in an expression is determined by the nature of the object, and not by the language. Therefore, little is said in the language specification about which commands and which operations in expressions are appropriate for a given object, or what the result is when an object is acted upon. This kind of information is found in Part C of this document for the so-called *standard objects* included in every CIL implementation.

Each object that can be manipulated through the CIL belongs to an *object class*.. An object class is a category that specifies the nature of an object and its behavior. Examples are PUMP, VALVE, and TEXT STRING. The classes of objects are typically inter-related in some way. Many object-oriented languages relate object classes by arranging them into a hierarchy, with the most general classes at the top and more specific classes beneath them. For example, an object class PUMP might be established to represent the actions and attributes common to all pumps. Beneath class PUMP, subclasses can be defined — like VACUUM PUMP and WATER PUMP — to represent unique kinds of pumps. A hierarchy of object classes has the virtue of simplicity: it is easy to determine how a class relates to other classes. While other means of interlinking object classes are possible, they result in greater complexity and it is not clear that the greater complexity would be advantageous for the CIL. Therefore, the CIL object model is based upon a hierarchical arrangement of classes.

A powerful part of the object-oriented approach is the concept of inheritance. Inheritance provides for the actions and attributes of one class of object to be passed on to its subclasses. For example, actions and attributes defined for class PUMP are inherited by its subclasses, like VACUUM PUMP and WATER PUMP. The inherited set of actions and attributes can be augmented by any additional actions or attributes that are unique to a class. For example, class VACUUM PUMP can define any actions or attributes that are unique to vacuum pumps. Because of inheritance, all pumps will share the attributes defined for class PUMP: if TURN ON is an action defined for class PUMP, then every pump in the system, including all vacuum pumps and water pumps, can be activated by a CIL command that invokes the action named TURN ON.

1.3.1. OBJECT CLASSES

An object-oriented language must have access to the interface agreements for all the classes of objects that it manipulates. Typically these interface agreements are stored in a database called an *object dictionary*. An interface agreement contained within an object dictionary is called a *class definition*. There may be more than one object dictionary. For example, there may be a project object dictionary and a user-specific object dictionary that augments it.

For the CIL, each object class definition provides:

- The name of the class.
- The class's parent in the class hierarchy. This is used to determine any inherited actions or attributes.
- A description of each attribute defined for the class. An attribute is an important characteristic of an object — for example, the length of a text string, or the speed of a pump — that can be sensed and sometimes modified using the CIL. The CIL does not care whether the value of an attribute is stored and then retrieved when needed or whether it is computed upon demand. Thus the concept of attributes in the CIL overlaps with the concept of functions in other languages (with an attribute being equivalent to a function's result).
- A formal description of each action that can be performed upon objects of the class. There are two kinds of actions supported by the CIL: operators and commands. Operators implement the actions that can be applied in expressions, like addition, multiplication, and so on. Commands are imperative statements, like OPEN THE POD BAY DOOR.

A format for object dictionaries is not specified in this document, nor is the CIL limited to any particular implementation of an object dictionary.

Two important characteristics often associated with object-oriented languages are present in the Control Interface object model: polymorphism; and overloading:

- Polymorphism — Even though a subclass may inherit actions from its parent class, the way in which the action is implemented may be different for the subclass. As an example, a system may use two different kinds of pumps: ACME pumps and AJAX pumps. The two kinds of pumps perform the same function but ACME pumps use a 28-volt pulse to turn on the pump while AJAX pumps require a digital string of bits to achieve the same action. To the user, the command to turn on both kinds of pumps can be the same — TURN ON PUMP — regardless of whether the specified pump is an ACME pump or an AJAX pump. The implementation of the TURN ON action for the class of ACME pumps is different from AJAX pumps, but that is transparent to the user. Thus polymorphism promotes and preserves a consistency in the actions of like classes while allowing for the actions to be implemented in unique ways for each class.
- Overloading — Actions may have more than one meaning assigned to them within a single class. For example, the multiplication operation for integer numbers allows

integer numbers to be multiplied by an integer (yielding an integer result) or by a real number (yielding a real number result). Even though the definition of integer multiplication is overloaded, the CIL can determine which of the two definitions to use in a given expression.

1.3.2. OBJECT INSTANCES

To the CIL, the world consists of a collection of objects. Every object instance belongs to on particular object class. The way in which the CIL acts upon an object, and the way in which the CIL expects the object to behave, is determined by the class.

In many cases, a user will require access to only a very small subset of the objects within a system. For example, a payload engineer may have permission to turn on and turn off a pump within his or her payload, but he or she must not be allowed to turn on or turn off a pump in a core spacecraft system. A lead spacecraft engineer may, on the other hand, be allowed to manipulate both payloads and core systems. This document assumes that there is a database — called an *object directory* — that names the objects that make up a system and that defines who has permission to access each object. There may be several distributed object directories.

Most objects that will be manipulated by the CIL will be identified by names registered in an object directory. For example, a particular vacuum pump onboard a spacecraft might have a name like PURGE PUMP #2 and this name would be entered into an object directory. Each entry in an object directory should also specify the class of the object. For example, the entry for PURGE PUMP #2 would indicate that it belongs to the object class VACUUM PUMP. When a CIL command like

TURN ON PURGE PUMP #2 ;

is processed, the CIL processor can locate the object PURGE PUMP #2 in an object directory, determine from the directory entry that this object is of class VACUUM PUMP, and then consult an object dictionary to determine that the action TURN ON is indeed allowed for vacuum pumps and hence for PURGE PUMP #2.

Object dictionaries and object directories can be implemented in many ways, Sometimes object dictionaries and object directories are combined into a single database called an *object encyclopedia.* This document does not require or constrain object dictionaries, directories or encyclopedias to be implemented in any particular way.

1.3.3. ABOUT THE STANDARD OBJECTS

All objects discussed in this specification derive from a primordial class called OBJECT. Beneath class OBJECT is a predefined set of objects called the Standard Objects. These object classes must be defined for every CIL implementation, although the implementation of these classes is not constrained by this document. A class called STANDARD OBJECT is defined to serve as the anchor for this set of classes. Attached to class

STANDARD OBJECT is a set of nine *base* classes. With one exception, objects that belong to the base classes all have a literal or explicit representation defined within the language specification.

Some of the base classes have standard subclasses. The class STRING is divided into several subclasses that are present in every CIL implementation. The main subclasses allow strings to represent text strings, bit strings and date-times. There are three subclasses of bit strings to represent numbers in binary, octal, and hexadecimal notation.

Objects of class MEASUREMENT hold real numbers, always with a unit of measurement specified. There are two standard measurement subclasses to represent real numbers and intervals of time. Other measurement classes can be added within an implementation to represent lengths, voltages, angles, and so on.

An enumeration object represents a selection from a list of choices. There are four standard enumeration subclasses:

See Part C of this document for the definition of each of the classes shown above and for the rules governing the addition of new object classes.

1.4. CIL PROCESSING

Some CIL statements can be entered by a user via a keyboard or potentially through a voice recognition system for immediate execution. This kind of execution is called *interpretive* execution and it is carried out using software that is called an *interpreter*. CIL interpreters may serve as an interface to spacecraft hardware and embedded software, or to other software systems and applications programs that don't have their own user interface. When a CIL command is entered by a user, the interpreter will verify that the syntax and semantics of the command are correct and will then carry out the command by passing the command request to one or more *backend processors* that will format the command (for example, into an SMS message) so that it can be interpreted correctly at its destination. The command message will then be routed to its destination using an appropriate transport mechanism.

CIL code may be compiled into procedures and saved. Later the compiled procedures can be retrieved and executed. A program designed to execute CIL code is an *executor*. Since there are several kinds of CIL statements that are valid only in procedures, an executor is typically more complex than an interpreter. However, an executor carries out commands in the same way as does an interpreter: when a command is to be executed, it is passed to an appropriate backend processor.

The following sections describe the process of parsing and executing CIL statements and procedures. These sections are not meant to restrict or constrain CIL processing, but rather to provide the reader with a background on CIL processing that can make the CIL specification easier to understand.

1.4.1. SYNTACTIC ANALYSIS

The syntactic analysis function determines whether or not the structure of a CIL statement is complete and grammatically correct. For example, a statement like

```
LET X := 2 +
```

is incorrect because the addition operator (+) requires two operands. The output of syntactic analysis is typically a data structure like a parse tree that conveys the relationships among the various grammatical elements of the statement. The syntax described for the CIL in this document is based on a context-free grammar. Most programming languages are built on context-free grammars and software tools for building parsers for these grammars are widely available.

1.4.2. SEMANTIC ANALYSIS

Semantic analysis determines whether or not a statement makes sense. Semantic errors in the CIL include:

- References to objects that don't exist or that are inaccessible to the user.

- Invoking an action that doesn't apply for a specific object. For example, a command like

  Turn Off DOOR1 ;

  doesn't make sense if the only actions defined for doors are OPEN and CLOSE. Another example is an attempt to add an integer number and a pump: addition is defined for numbers but not for pumps.

- References to attributes that an object does not possess; for example, the command

  Set LENGTH of PUMP2 To 10 ;

  refers to an attribute named LENGTH that a pump would not possess.

Semantic analysis of CIL will rely heavily upon the information supplied in object dictionaries and object directories. The output of semantic analysis is typically a data structure like an attributed parse tree which has the same basic form as the syntax parse tree but with additional information added to convey semantic information.

1.4.3. TRANSLATION

The data structure generated by semantic analysis (like an attributed parse tree) is usually translated into some form of code that can be executed more readily. Most compiled languages are translated into the machine language for a specific target computer. An alternative is to generate machine-independent code that can be efficiently interpreted on many different target computers. The latter approach is widely used with test and operations languages and is recommended for the CIL. There is no requirement for the translated code generated in interpretive mode and compiled mode to be the same, but this may simplify the software needed for the execution function.

1.4.4. EXECUTION

In interpretive mode, a CIL statement is executed immediately after analysis and translation. In compiled mode, the translated code of a CIL procedure may be read in from storage and executed. A statement is executed either by directly performing the action specified in the statement or by passing control to a backend processor for further processing. The former is the case for statements like assignment statements, and repeat loops. The latter is the case for commands sent to external objects like pumps, etc. Many implementations of the CIL will be required to log statements as they are executed.

1.4.5. BACKEND PROCESSING

When a command is issued to turn on a vacuum pump, somewhere outside of the CIL interpreter or executor there must be a piece of software that has cognizance of and control over the vacuum pump we want to turn on. To affect the vacuum pump, the CIL execution function must inform the software that controls the pump that the pump is to be turned on. The external programs that carry out actions specified by UIL statements are called backend processors because they are at the tail of the CIL processing chain.

1.4.6. BINDING OF RESOURCES

A key issue regarding compiled CIL code is how to bind together multiple procedures required for a single task. For example, if a test procedure calls another procedure when a fault is detected, then these two procedures should perhaps be linked together after compilation so that they are both available when the test is performed. The mechanism for doing this is not addressed in this document.

Similarly, a mechanism is needed for binding CIL procedures to the object dictionaries, directories and encyclopedias that they need. Again this document does not specify or restrict how this is done.

2. LANGUAGE DEFINITION

2.1. ABOUT THE LANGUAGE DEFINITION

This is the formal specification for the syntax and semantics of the Control Interface Language. The character set of the CIL is defined first, followed by the lexical rules that govern how characters are strung together to form meaningful lexical elements. A formal context-free syntax, specified using a Backus-Naur Form (BNF) notation, is given for all CIL statements. The narrative that describes each syntax rule also presents additional rules that govern the semantical interpretation of the language.

2.1.1. NOTATION

The CIL is described herein using Backus-Naur Form (BNF) notation. Language elements are defined using productions of the form

defined_element ::= definition

where the definition is composed from the following components:

a) Lower case words, some containing underscores, are used to denote syntactic categories. For example:

    adding_operator

   Whenever the name of a syntactic category is used outside of the formal BNF specification, spaces take the place of underscores (for example, adding operator).

b) Boldface type is used to denote reserved words. For example:

    return

   Special characters used as syntactic elements also appear in boldface type.

c) Square brackets enclose optional elements. The elements within the brackets may occur once at most.

d) Braces enclose repeated elements. The elements within the braces may appear zero, one or more times. The repetitions occur from left to right.

e) A vertical bar separates alternative elements.

f) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *object*_name and *units*_name are equivalent to name; *object*_name is used in places where the name of an object like PUMP1 is required and *units*_name where the name of some engineering unit like VOLTS is expected.

In the examples of CIL found in this document, the following conventions are used to make the examples easier to read:

a) Reserved words used to introduce and terminate statements and to mark major clauses within statements are shown with the first letter of each word capitalized:

```
If . . .
     Then . . .
     Else . . .
End If ;
```

Words (usually verbs and adverbs) specifying actions within commands also follow this convention:

Perform

Turn On

b) The names used to identify objects, attributes, classes and parameters are shown in capital letters:

PUMP

CHECK VALVE

Words appearing within literal values, including symbolic literals and units within measurement literals, are also given in all capital letters:

OPEN

RPM

c) Reserved words appearing as operators are shown in lower case:

of

is not

Here are some sample statements:

Perform PUMP TEST With TEST DURATION := 14 MINUTES 30 SECS ;

```
Set SPEED of VACUUM PUMP To 1234 RPM ;
If PUMP is ON and FLUSH VALVE is not OPEN
  Then
          Print PUMP ALARM MESSAGE ;
  End If ;
```

2.2. CHARACTER SET

CIL statements and procedures are composed from the ISO 646 character set. ASCII is the U.S. version of ISO 646; in other countries, certain symbols have a different graphical representation. The ISO 646 character set is a subset of the ISO 8859-1 Latin-1 character set, which in turn is a subset of the newer ISO/IEC 10646 and Unicode character sets. Any of these character sets may be used for a CIL application, since the language and the behavior of base objects as defined in this specficiation do not depend upon the number of bits needed to represent a character, the binary values assigned to characters within a code, or the collating sequence of characters. The only difference that comes from using one of the newer character sets is that the category of “other characters” as defined below may be larger than for the ISO 646 character set.

```
character ::=  basic_character |  spacing_character |  formatting_character  |
         other_ character

basic_character ::= letter  |  digit  |  special_character
spacing_character ::=       space |  horizontal_tab
formatting_character ::=  carriage_return  |  line_feed  |  vertical_tab
```

The basic character set for the CIL is sufficient for writing any CIL statement or procedure. The basic character set is as follows:

- Letters are A - Z and a - z. The CIL is case insensitive, meaning that lower case letters are treated as identical to their upper-case counterparts. Thus the following statements are equivalent:

  TURN OFF PUMP1 ;

  Turn Off Pump1 ;

  turn off pump1 ;

  An exception to the case rule is when lower-case letters appear as part of text string objects. For example, the text string "abc" is not be the same as the string "ABC".

- Digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

- Special characters, used for such things as mathematical operators in expressions, are as follows (the sections of this document which contain further information on the lexical usage of each character are cited in parentheses):

  ; The semi-colon character is used to mark the end of a statement.

  " The quotation mark character is used to mark the beginning and end of text string literals. For example, "All systems are go" (Section B-3.3).

(and **)** Parentheses is used to enclose lists and to set off parts of an expression (Section B-3.4).

. The period, or dot, is used as a decimal point in floating point constants (Section B-3.2). Two consecutive dots form the operator indicating a range of values (Section B-3.4).

< The left angle bracket is the *less than* operator in relational expressions. Two consecutive left angle brackets serve as the opening delimiter for a units expressions (Section B-3.4).

> The right angle bracket is the *greater than* operator in relational expressions. Two consecutive right angle brackets serve as the closing delimiter for units expressions (Section B-3.4).

+ The plus sign is the addition operator in arithmetic expressions (Section B-3.4).

– The dash, or minus sign, is the subtraction operator and the negation operator in arithmetic expressions. Two dashes in a row mark the beginning of a comment (Section B-3.4).

***** The asterisk is the multiplication operator in arithmetic expressions. Two asterisks in a row are used for the exponentiation operator (Section B-3.4).

/ The slash is the division operator in arithmetic expressions and part of the *not equal to* operator in relational expressions (Section B-3.4). It is also used within DT (date-time) strings to set off the components of a date.

= The equals sign is used as part of the assignment operator and as the *equal to* operator in relational expressions (Section B-3.4).

: The colon is used as a delimiter within declarations and Step statements (Section B-3.4). It is also used within DT (date-time) strings to set off the components of a clock time.

, The comma is used to separate the items of a list (Section B-3.4).

& The ampersand is the concatenation operator (Section B-3.4).

The pound (number) sign is used to introduce words that start with digits (Section B-3.1).

The spacing characters — space and horizontal tab — are separators between lexical elements and are used to insert white space into CIL statements.

The formatting characters — carriage return, line feed, and vertical tab — are also separators between lexical elements and are used to organize CIL statements into lines.

The other characters are the graphic (printing) characters that are not in the basic character set. These characters are not required to write CIL statements but are allowed within strings. For the ISO 646 character set, the graphic characters that fall into this category are:

    ! $ % @ ? [\ ] ^ ` { | } ~

2.3. LEXICAL ELEMENTS

Lexical elements are the building blocks of a language. CIL statements and procedures are composed by stringing lexical elements together according to the grammatical rules presented later. The CIL has the following lexical elements:

- Words
- Numbers
- Strings
- Special symbols used for operators, etc.

2.3.1. WORDS

Words are used to represent names, units of measurement, symbol values, and more. Words are composed of alphabetic characters and digits as follows:

word ::= letter {letter | digit} | **#** digit {letter | digit}

The concept of a word as used here is broader than the normal meaning for natural languages and encompasses abbreviations and mnemonics.

<u>Examples</u>

PUMP

C3PO

CLOSE

AMPERES

AMPS

#301J

Most words begin with a letter. If a word begins with a digit, it must be preceded by the # character (which is pronounced *number).*

2.3.1.1. Reserved Words

Certain words are reserved to introduce or set off the different parts of CIL statements and procedures. These reserved words cannot be used for other purposes.

after	**all**	**and**	**at**	**before**
begin	**by**	**choice**	**constant**	**declare**
derive	**else**	**end**	**every**	**exit for**
from	**goto**	**if**	**into**	**is let**

not	**of**	**or**	**procedure**	**repeat**
return	**stop**	**the**	**then**	**to until**
wait	**when**	**where**	**while**	**with**
within				

2.3.1.2. Modifiers

The following words are permitted as adverbs that modify the verb in commands. Modifiers are not allowed within names, to avoid amibiguity when parsing commands and other statements. They are allowed in symbols.

modifier ::= ON | OFF | IN | OUT | UP | DOWN

2.3.2. NUMBERS

There are two kinds of lexical elements that specify numeric values: integer and real. Integer numbers are represented using decimal notation (like 189). Real numbers are represented in simple decimal notation (like 123.4) or in a scientific notation that includes a base-ten exponent (for example, 1.234E2).

Note that the definitions for numeric lexical elements given below do not provide for an explicit plus or minus sign. This is because signed numbers are handled as an expression.

2.3.2.1. Integer Numbers

The format for integer numbers is:

integer ::= digit {digit}

<u>Examples of Integer Numbers</u>

0

4095

123456789

2.3.2.2. Real Numbers

The format for real numbers is:

real ::= simple_real [exponent]

simple_real ::= integer **.** [integer] | **.** integer

exponent ::= **E** [+] integer | **E -** integer

If an exponent appears in a real number, the letter E is allowed in either upper or lower case. If the sign does not appear in a exponent, the exponent is presumed to be positive.

Examples of Real Numbers

0.

1234.567

.003141593E3

1.415E-10

2.3.3. STRINGS

A string lexical element begins and ends with a quotation mark and contains a string of one or more characters.

string ::= **"** {basic_character | spacing_character | other_character} **"**

A string may contain any of the basic characters (except for the quotation mark), spacing characters or other characters. Strings may not contain formatting characters, which means that a string must be contained on a single line. Null strings are allowed and are represented as two quotation marks without any intervening characters.

2.3.4. SPECIAL CHARACTERS

Some special characters appear as part of the lexical elements described above. Examples include the decimal point within a real number and the quotation marks that delimit a text string. Often, however, a special character or a two-character sequence stands on its own as a lexical element — usually as an operator within an expression.

The following special characters are considered lexical elements unless they appear within a character string or as part of one of the lexical elements described above:

+	*Unary plus* operator or *add* operator
-	*Unary minus* operator or *subtract* operator
*****	*Multiply* operator
/	*Divide* operator
&	*Concatenation* operator
<	*Less than* relational operator
>	*Greater than* relational operator
=	*Equals* relational operator
(	Opening delimiter for a list
)	Closing delimiter for a list
,	Delimiter between the elements of a list
:	Delimiter in declarations and Step statements

;	Semi-colons mark the end of a statement

The following two-character sequences are lexical elements unless they appear within a character string:

:=	*Assignment* operator
/=	*Not equal* relational operator
>=	*Greater than or equal* relational operator
<=	*Less than or equal* relational operator
**	*Exponentiation* arithmetic operator
..	*Range* operator for indicating a range of values
<<	Opening delimiter for a units expression
>>	Closing delimiter for a units expression
--	Two dash characters indicate the start of a comment

2.3.5. SEPARATING LEXICAL ELEMENTS

A lexical element is terminated by the first character that is not a part of the lexical element. For example, in the expression 2.0*PI, even though the characters run together, there are three distinct lexical elements: the real number 2.0, the asterisk serving as multiplication operator, and the word PI.

In some cases there can be ambiguity about where lexical elements end. For example, is ** an exponentiation operator or two multiply operators? In such cases, the longest lexical element that can be formed is chosen. Thus two asterisks in succession are recognized as an exponentiation operator.

In some cases spacing characters must be inserted to separate lexical elements (for example, between two consecutive words). Spacing or formatting characters are allowed before or after any lexical element and between any pair of lexical elements without changing the meaning of a statement. Two or more adjacent spacing or formatting characters have the same effect as a single such character.

2.3.6. LINES AND STATEMENTS

Lexical elements are strung together to form statements and procedures. A statement or procedure may occupy part of one line, an entire line, or more than one line. A line is defined as a string of characters ending with a formatting character (or a string of consecutive formatting characters). Lexical elements cannot cross lines. Put another way, formatting characters are not allowed within lexical elements. Blank lines — lines consisting of no characters or only of spacing characters — are allowed and do not change the meaning of statements and procedures.

A line may contain a comment. A comment is introduced by a pair of adjacent dash characters. All characters on the line from the comment indicator up to but not including the formatting character that terminates the line are treated as part of the comment. Comments are ignored, and the presence (or absence) of a comment never changes the meaning of a statement or procedure. An example of some CIL statements with comments follows:

```
If PUMP1 is ON
   Then
       Turn Off PUMP1 ;      -- Only one pump can be on at a time
 End If;
 Turn On PUMP2 ;             -- Start the other pump
 Perform PUMP TEST With Pump := PUMP2,    -- Check out the pump
                               Max Speed := 1000 RPM ;
```

Note that in the example above, some of the comments appear at the end of statements and others within statements.

2.4. BASIC SYNTACTIC COMPONENTS

This section describes basic language structures built from lexical elements and appearing in many different types of CIL statements, including:

- Names
- Literals
- Ranges and Lists
- Expressions
- Assignment Operations

2.4.1. NAMES

Names are used to identify various entities within the CIL, including objects, object classes, attributes of objects, procedures, steps, and units of measure-ment. CIL names are composed of one or more words:

name ::= word {word}

Spacing or formatting characters between the words within a name have no significance when interpreting the name. Therefore the names PARTS DATA BASE and PARTS DATABASE are equivalent.

A name must not contain any of the reserved words listed in Section B-3.1. For example, the name EXIT DOOR LATCH is not legal since it contains the reserved word *exit*. (If the reserved word is omitted, the name DOOR LATCH is legal.)

There is no limit on the length of a name, except for any implementation-specific limit on the length of a statement.

<u>Examples of Names for Object Classes</u>

INTEGER
VALVE

VACUUM PUMP

ACME #301J VACUUM PUMP

<u>Examples of Names for Objects</u>

POD BAY DOOR

H2O PUMP #1

AOS TIME

Examples of Names for Units of Measurement

SECONDS
SQUARE FEET
KM PER SEC

2.4.1.1. Using Names

The same name may be used for more than one purpose without causing ambiguity. A specific name may appear as one or more of the following:

- Class name
- Object name
- Attribute name
- Action name in a command
- A unit of measurement
- Step name
- Procedure name

For example, there is a standard object class named RANGE and there is also an attribute named RANGE that is defined for the standard object class INTEGER, but this creates no ambiguity in the language because it is always possible to differentiate between the name of a class and the name of an attribute.

2.4.1.2. Object Names

Not all objects have names. Literals, described in Section B-4.2, are objects and they have a value, but they don't have a name. An expression can manipulate literals directly without having to refer to them by name, as in the expression 3+2, which adds two objects of class INTEGER. Nonetheless, most objects are referred to by name.

Before an external object — like a pump or an robot arm — can be manipulated by a CIL statement, the object must be given a name. The names of external objects are assumed to exist within a global environment in which the CIL is running, although how these names become available to the language is not specified here. Names can also be created during an interactive CIL session or within a procedure by using the name declaration statement discussed in Section B-5.1. A name declaration can bind a name to an object — this is called a constant declaration — or it can create a name that may be assigned to an object and then reassigned to another object at a later time (this is called a variable name). Variable names can be assigned to objects using the assignment operation. The assignment operation is described in Section B-4.5. An object may have more than one name assigned to it at any given time. This document does not specify what happens to objects that are no longer referenced by any name. Such objects may be deleted and any storage space required to represent them reclaimed.

Reserved Object Names

There are three special objects that are part of the CIL and their names are reserved (no other object or variable can bear these names):

- NOTHING — This is the object to which an object reference points when it points to no other object. It is the object pointed to by an uniitialized variable. If this object is referenced within an expression or command an error will result. See Section B-4.5 for further information on the use of this name.
- RESULT — An object that contains the result of the most recent Get command. See Section B-5.4 for further information on this name.
- IT — A synonym for RESULT.

2.4.1.3. Attribute Names

An object class may define attributes that specify important characteristics of objects of the class. Each attribute of an object has a name. For example, the standard object class TEXT defines an attribute named LENGTH to indicate the length of a text string. Attribute names always appear within CIL statements in association with one or more objects. The association between objects and attributes is specified using the *of* operator. For example, if the name Y AXIS TITLE is an object of class TEXT, then LENGTH of Y AXIS TITLE refers to the number of characters in the title string.

An attribute with the same name may be defined for more than one object class and the meaning of the attribute may differ according to its class. For example, the attribute LENGTH is defined for both object class TEXT and object class BIT STRING; for text objects the length is measured in characters, while for bit strings length is measured in bits.

2.4.2. LITERALS

A literal is an explicit representation of an object. The CIL has literal representations for objects of the following classes:

- INTEGER
- MEASUREMENT
- STRING
- SYMBOL

literal ::= integer_literal | measurement_literal | string_literal | symbolic_literal

Literals are objects and each literal value is subject to any implementation-specific limitations or restrictions that apply to its class. The lexical elements described in the previous section are simply strings of characters and any particular lexical element will look and act the same way on every computer on which CIL is compiled or interpreted. Literals, on the other hand, may not always act the same since the way in which objects are implemented and stored can vary from computer to computer, giving rise to differences in the size, range and accuracy of the objects that can be represented. For example, while there is no limit on the number of digits in a numeric lexical element, the representation of that number as a literal value is subject to a computer's limitations on range and precision for numeric objects. Literal values are also subject to semantic limitations that do not apply to lexical elements.

2.4.2.1. Integer Literals

Integer literals represent the values of objects of the numeric class INTEGER. Formal definition of this class is found in Section C-2.3. The format of an integer literal is:

integer_literal ::= integer

The value of an integer literal must be within the range specified by the attribute RANGE for class INTEGER.

<u>Examples of Integer Literals</u>

0

1234
14471210

2.4.2.2. Measurement Literals

Most computer languages have a literal representation of the value of real numbers. Usually these values represent some kind of measurement, but the kind of measurement — for example, distance, volume or voltage — is not explicit. In CIL the kind of measurement is always explicit; hence all real numbers belong to the class MEASUREMENT. Measurement literals represent objects of class MEASUREMENT and its subclasses. Only two measurement subclasses — REAL and TIME INTERVAL — are mandatory in the standard object set. Other subclasses can be added to accomodate distances, voltages, etc. The formal definitions of class MEASUREMENT and its subclasses are found in Sections C-2.4 through C-2.6.

Measurement literals may consist of a single measurement value. For example, 5.11 VOLTS. Measurement literals may also consist of two or more measurement values. Two examples are:

2 FEET 6.5 INCHES

1 DAY 12 HOURS 30 MINUTES

Sometimes, particularly in scientific applications, complicated units of measure arise, and for those cases an alternative equation form for specifying units is provided. An example is:

16.35 <<VOLTS / METER**2 / SEC>>

The format of a measurement literal is:

measurement_literal ::= measurement {measurement}

measurement ::= integer units | real [units]

units ::= *units*_name | << units_expression >>
units_expression ::= units_factor {multiplying_operator units_factor}
units_factor ::= *units*_name [exponentiation_operator units_exponent]
units_exponent ::= [+] integer | **-** integer
multiplying_operator ::= ***** | **/**
exponentiation_operator ::= ******

A measurement literal may consist solely of a real number without units specified, in which case the value is assigned to the measurement subclass REAL. If a unit of measure is specfied, then the measurement literal is classified into one of the subclasses of class MEASUREMENT by comparing its units to the known units for each subclass. For example, a measurement expressed in feet or meters will be classified as an object belonging to the measurement class LENGTH while a measurement expressed in grams or pounds will be assigned to class MASS.

If a measurement literal consists of two or more measurements, each of the measurements in the literal must belong to the same measurement class. For example, 3 FEET 6 INCHES is legal; 3 FEET 6 SECONDS is not. The measurements in a measurement literal are allowed to appear in any order. For example, 6 INCHES 3 FEET is equivalent to 3 FEET 6 INCHES (although common usage clearly favors the latter format).

The multiplication, division and exponentiation operators for units expressions are the same as for arithmetic expressions. As with arithmetic expressions, exponentiation has a higher precedence than multiplication and division.

<u>Examples of Measurement Literals</u>

167.
1234 RPM
1.83E6 VOLTS

4.5 SQUARE METERS

1 MILE 1200 FEET

1 HR 12 MINS 14.6 SECS

11.57 <<FEET/SEC**2>>

2.4.2.3. Strings

String lterals represent objects that belong to a subtype of class STRING. The class STRING itself is a virtual class and there are no objects of the class: in practice all strings represent a specialization, belonging to one of the string subclasses. There are three string subclasses defined in the standard object set: TEXT for representing general text strings; BIT STRING for representing strings of bits; and DT strings for representing dates and times. Other kinds of string subclasses may be created. For example, a string subclassed called URL might be created to hold an Internet universal resource location. The semantics for different kinds of strings are different, but they share a common literal syntax:

string_literal ::= [*class*_name] string

The class name detemines how the string semantics will be interpreted. If the class name is omitted, the string is assigned to class TEXT.

Semantics of Text Strings

If a string literal appears without a preceding class name, or if the class name is TEXT, then the string is interpreted as a text string. Text strings follow the general rules for strings listed in Section B-3.3 above. The case of alphabetic characters is preserved. Additional semantics are defined to allow rudimentary text formatting when strings are output. Text formatting can be accomplished by using the following two-character sequences, the first character of which is always a backslash character:

\a Alarm: Upon output an alarm tone (bell, beep, etc.) is sounded.

\n Newline: Upon output the marker is replaced by a newline sequence (an implementation-specific sequence of formatting characters).

\t Tab: Upon output the \t is replaced by a horizontal tab character.

\\ Backslash: Two contiguous backslashes indicate that a single backslash character is to appear in the output stream.

The length of a text string is measured in characters.

<u>Examples of Text Strings</u>

"This is a string"

"He said ‘How do you do!’ to me"

“\aWarning: Safing Circuit is Offline \n”

The second example shows that apostrophes can serve to set off quotations within strings, thus getting around the proscription against having quotation marks within a string. The third example demonstrates the use of special formatting sequences: the alarm sequence at the beginning of the string; and the newline sequence at the end of the string.

Semantics of Bit Strings

Bit string objects can hold binary-encoded data other than numbers. For example, each bit of a bit string might represent a binary flag indicating that a particular circuit is off or on.

Each (non-spacing) character of a bit string is a digit in a specific number base. Three number bases — binary, octal and hexadecimal — are provided through standard subclasses of class BIT STRING (see Section C-2.9 for the formal definition of BIT STRING). The main purpose of these subclasses is to make it easier to represent bit string literals and bit string output in the form that is most meaningful to users.

Class	Number Base	Characters Allowed as Digits
B	Binary	0, 1
O	Octal	0, 1, 2, 3, 4, 5, 6, 7
X	Hexadecimal	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

For the digits of hexadecimal literals represented by a letter, the upper case and corresponding lower case letters (for example, D and d) are treated as identical.

Spacing characters may appear before the first digit, after the last digit or between digits of a bit string. These characters are ignored. Thus the bit strings B “0 1 0 1“, B “01 01”, and B “0101” are equivalent.

The length of a bit string is measured in bits. The length of a bit string literal is computed by multiplying the number of digits in the string by the number 1, 3, or 4 for binary, octal and hexadecimal strings, respectively.

<u>Examples of Bit Strings</u>

B ”10110000”
O ”17364253”

X ” 4BF3 02a9 6445 cd70 ”

The last example above shows that blank characters can be used within a bit string to set the digits apart for better readability.

Semantics of DT Strings

DT strings represent calendar dates and clock times. A date-time string contains either a calendar date, specified according to the modern Gregorian calendar, or a clock time specified using a 24-hour clock, or both a date and time. Intervals of time — indicating, for example, that something is to occur one hour and 36 minutes from now — are usually not specified with DT literals; rather they are specified using objects of the measurement class TIME INTERVAL.

Four different forms of dates are supported. Each of the date forms is composed from two or three of the following components. Four of the components are represented as integer numbers, following the same rules as for integer literals. The other component is a symbol drawn from a list of month names.

Component	Definition	Class and Range of Value
dom	Day of month	Integer , 1 .. 31 (depending upon month)
doy	Day of year	Integer, 1 .. 365 (366 in a leap year)
month	Month of year	Integer, 1 .. 12
moname	Month name	Month Name (a subclass of Enumeration)
year	Year	Integer , 1900 .. 2099

The month name is a symbol value drawn from the enumeration class Month Name, which is described in Section C-2.12. The maximum value for the day of month is determined by the month (and for the month of February, by whether or not it is a leap year). The four allowed date formats are shown below:

Date Format	Examples
year / doy	1998 / 080
year / month / dom	1998 / 3 / 21
year moname dom	1998 Mar 21
moname dom , year	March 21, 1998

Times are composed from the components listed in the following table:

Component	Description	Class and Range of Value
hour	Hour of day	Integer , 0 .. 23
min	Minute of hour	Integer, 0 .. 59
sec	Second of minute	Integer or Real, 0.0 <= sec < 60.0
sod	Seconds of day	Integer or Real, 0.0 <= sod < 86400.0
zone	Time zone indicator	Time Zone (a subclass of Enumeration)

The allowed time formats are:

Time Format	Examples
hour : min	14 : 30
hour : min : sec	14 : 30 : 45.7
sod	52245.70

A time zone indicator — a symbol drawn from the enumeration class TIME ZONE described in Section C-2.12 — is allowed after any time. For example:

14:30:45.7 EST

In the example above, the time is given relative to Eastern Standard Time. If a time zone indicator is not included, then the time is interpreted as Coordinated Universal Time (UTC).

A date or a time may appear by itself within a date-time string literal or the date and time may be combined. Any form of date may be combined with any form of time. The date and time portion must be separated by one or more spacing characters. A date and time may appear with either the date part first or the time part first. Thus the following two examples are equivalent:

DT “3/21/1998 12:30 UTC”
DT ”12:30 UTC 3/21/1998”

Spacing characters are allowed at the beginning or end of a date-time string or between the components of a date or time without changing the meaning of the date-time string.

Examples of DT Literals

DT “2001/1/1 00:00”

DT “1996/301 12:30:45.5 CST”

DT “1 Nov 1996 12:30:45.5 CST”

DT “August 21, 1999”
DT “16:00:00.0 MDT”

2.4.2.4. Symbolic Literals

Symbolic literals represent objects of class SYMBOL. The formal description of the class is contained in Section C-2.11. The format of a symbolic literal is:

symbol ::= [the] name | [the] extended_name

extended_name ::= {word} modifier {word | modifier}

Symbols express values using words. For example, the symbol JANUARY represents a month of the year. A symbol may be a name, like HIGH GAIN. Unlike names, however, symbols may also include any of the modifier words described in Section B-3.1. For example, the symbols ON and OFF may be used to indicate the state of a circuit and the status of a test may be represented by the symbol IN PROGRESS.

The reserved word *the* is allowed before a symbol. For example, in the command MOVE MANIPULATOR ARM To the LEFT, the word *the* appearing before the symbol LEFT makes the command more like natural language. If the word *the* appears before a symbol it is ignored and is not considered part of the name. Thus the sample command above has the same meaning regardless of whether the word *the* appears or not.

Symbols are often used as the value of enumeration objects. Enumeration objects belong to a subclass of the virtual class ENUMERATION. Each enumeration class has an attribute named CHOICES that is assigned to a list of objects; an enumeration object has one of these objects as its value. For example, an enumeration class named DEVICE STATE might define the choices ON, OFF and STANDBY and each object of class DEVICE STATE will have one of these three symbols for its value. See Section C-2.12 for the formal definition of class ENUMERATION.

An important enumeration subclass used by the CIL is class LOGICAL. All relational and membership operations in expressions produce an object of class LOGICAL. The choices for class LOGICAL are the symbols TRUE and FALSE. Section C-2.13 gives the formal definition of class LOGICAL.

<u>Examples of Symbolic Literals</u>

ON

FALSE

HIGH POWER SETTING

OFF SCALE

2.4.3. RANGES AND LISTS

The CIL provides explicit representations for objects belonging to the standard object classes RANGE and LIST. These explicit ranges and explicit lists differ somewhat from the literals of the previous section because their values are determined by evaluating expressions. The formal definitions of classes RANGE and LIST are found in Sections C-2.14 and C-2.15, respectively.

2.4.3.1. Ranges

Objects of class RANGE specify a range of numeric values. The format for an explicit range object is:

range ::= simple_expression range_operator simple_expression

range_operator ::= **..**

The explicit range L .. R specifies the values from L to R, inclusive. The values L and R are called the lower bound and the upper bound of the range. The L and R values are specified as simple expressions. A simple expression can be an object by itself or one or more objects in an arithmetic expression. For example, 10 is a simple expression, and so is INDEX, and so is INDEX+10. The formal definition of simple expressions is given in Section 4.4, below.

The simple expression to the left of the range operator determines the lower bound of the range and the simple expression to the right of the range operator determines the upper bound. Both the L and R values must belong to the same class and this must be class INTEGER or one of the measurement classes. Both the L and R values must be within the range of values that can be represented for that class, as determined by the attribute RANGE for the class. The value of the upper bound R must be greater than or equal to the lower bound L.

<u>Examples of Ranges</u>

0 ..10

60.0 .. 90.5

-5.0 VOLTS .. +5.0 VOLTS

INDEX+10 .. LENGTH-INDEX+1

2.4.3.2. Lists

Objects of class LIST hold collections of objects. The format of an explicit list is:

> list ::= object_list | attribute_list
> object_list ::= expression {**,** expression}
> attribute_list ::= *name*_list **of** list

An explicit list is converted into an object of class LIST that contains zero, one or more objects. The objects in a list are called the list's members. An object list is an explicit list where the members are determined by evaluating expressions. An attribute list is an explicit list that specifies one or more attributes of one or more objects.

Object Lists

Each member of an object list is represented by an expression. If an object list is encountered during execution of a statement, each expression is evaluated and the resultant objects become the members of the list. Any class of object is allowed in an object list. For example, if the list

> 9-4, OFF, 5.0*2.5

is encountered, the three expressions are evaluated and, assuming that OFF is a symbol, the result is the list

5, OFF, 12.5

Lists enclosed within parentheses are themselves expressions, and an object list is allowed to have a list for any member, provided that the member list is enclosed in parentheses. There is no limit to the depth to which lists may be nested within lists. As an example of lists within a list, the list

(1,2) , (3,4) , 5, 6

has four members, the first two of which are lists and the last two of which are integer objects.

Examples of Object Lists

1, 2, 3, 4, 5

PUMP1, PUMP2, PUMP3

(1.1, 1.2, 1.3), (2.1, 2.2, 2.3), (3.1, 3.2, 3.3)

(PUMP1, "Main Pump", ON), (PUMP2, "Auxiliary Pump", OFF)

Attribute Lists

Attribute lists are explicit lists used to specify the attributes of objects. The most common attribute list consists of a list of names of one or more attributes to the left of an *of* operator and an object list to the right of the *of* operator. For example, the attribute list

SPEED of PUMP1, PUMP2

specifies a single attribute named SPEED and two objects named PUMP1 and PUMP2. The objects in the object list need not all belong to the same object class; however, each of the names in the name list must be the name of an attribute defined for each of the object classes represented in the object list. For example, the attribute list

LENGTH of B”101”, "ABCDE"

is allowed since the attribute LENGTH is defined for both bit strings and text strings. The result of this attribute list is 3, 5 with the first member being the length of the bit string (in bits) and the second member being the length of the text string (in characters).

The format of an attribute list permits all of the following:

- Specification of a single attribute of a single object. For example:
  TEMPERATURE of LAB MODULE
- Specification of a single attribute of multiple objects:

SPEED of PUMP1, PUMP2, PUMP3

- Specification of multiple attributes of a single object or multiple objects:

  SPEED, TEMPERATURE of PUMP1, PUMP2

When an attribute list is encountered in a statement, it is converted into an object of class LIST by using the following rules:

1) First produce a list that has one member for each member of the object list found on the right side of the *of* operator. Each member of this list is an attribute list that includes all of the attributes listed to the left of the *of* operator, but for only a single object. The order of the objects in the original attribute list is preserved.

2) If more than one attribute is specified in the attribute list, convert each member of the list produced through rule 1 into a list whose members are attribute lists that specify a single attribute of a single object. The order of the attributes in the original attribute list is preserved.

For example, assume that there are two storage tanks named TANK1 and TANK2 and the object class to which they belong defines two attributes TEMP and PRESSURE.

An attribute list that specifies the temperature and pressure of both tanks looks like:

TEMP, PRESSURE of TANK1, TANK2

Applying rule 1 converts this into a list with two members, with each member being an attribute list for a single object:

(TEMP, PRESSURE of TANK1) , (TEMP, PRESSURE of TANK2)

Applying rule 2 divides each member of the list directly above into two members, each of which specifies a single attribute of a single object:

((TEMP of TANK1) , (PRESSURE of TANK1)) ,
((TEMP of TANK2) , (PRESSURE of TANK2))

An attribute list may also specify an attribute of an attribute of an object. For example, RANGE of TEMP of TANK1. The *of* operator is right-associative, meaning that the rightmost attribute list is evaluated first. For example, RANGE of TEMP of TANK1 evaluates as RANGE of (TEMP of TANK1), thereby referring to the range of values that the attribute TEMP may have for the object TANK1. Chained attribute lists are converted into object lists using the two rules given above. For example, the attribute list:

RANGE of TEMP, PRESSURE of TANK1, TANK2

reduces to the list:

> ((RANGE of TEMP of TANK1) , (RANGE of PRESSURE of TANK1)) ,
> ((RANGE of TEMP of TANK2) , (RANGE of PRESSURE of TANK2))

2.4.4. EXPRESSIONS

An expression is a formula that defines the computation of a value. Objects are the *operands* within expressions — they are what is acted upon. *Operators* are special symbols that specify some action that is to be applied to one or two operands. Each operation produces an object as its result. The format for CIL expressions is:

expression ::=
relation |
relation **and** relation {**and** relation} |
relation **or** relation {**or** relation}

relation ::= extent |
not extent |
extent relational_operator extent |
extent membership_operator extent

extent ::= simple_expression |
range

simple_expression ::=
[unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [exponentiation_operator primary]

primary ::= literal |
[the] *object*_name |
[name] **(** [list] **)**

relational_operator ::= **is** | **is not** | **=** | **/=** | **<** | **<=** | **>** | **>=**

membership_operator ::= **is within** | **is not within**

unary_adding_operator ::= **+** | **-**

binary_adding_operator ::= **+** | **-** | **&**

multiplying_operator ::= ***** | **/**

exponentiation_operator ::= ******

2.4.4.1. Operators and Their Precedence

The syntax rules for expressions establish the relative precedence of operators. The expression operators have the following relative precedence, listed from lowest to highest:

- The logical connecting operators *and* and *or*;
- The logical not operator, the relational operators and the membership operators;
- The adding operators (both unary and binary);
- The multiplying operators;
- The exponentiation operator.

The syntax rules for expressions do not specify which operators are valid for each class of operand, nor do they specify the result of an operation. An expression that attempts to multiply a number and a pump object together may be syntactically correct; but of course multiplication will likely not be a defined operation for a pump and an error will be reported during semantic analysis of the expression. The legal operations that can be performed on an object are specified in the formal definition of its object class. Each object class definition specifies:

- The operators that may be applied to objects of the class;
- For binary operations, the object class or classes allowed for the second operand;
- For each defined operation, the object class of the result of the operation;
- A description of what happens if the operation fails during execution.

Since the class of object resulting from each operation is defined in the object class definitions for the operations, it is always possible to determine during semantic analysis the class of object that will result from an expression. The class of the first operand determines which class definition to consult. For example, in the expression 3+2.0, the first operand is an integer object and so the definition of class INTEGER will be searched for an action that permits addition of an integer number and an object of class REAL. If an operation is not defined for the first operand's object class, then an error is reported during semantic analysis. For example, the expression "ABC"+3 will result in an error since the action of addition is not defined for text strings. For a binary operation, the object class of the second operand must be allowed for the operation or an error is reported during semantic analysis. For example, the expression 3+"ABC" will result in an error even though addition is defined for integers, because the action of addition as defined for class INTEGER does not permit a text string as the second operand.

In a sequence of operators of the same precedence level, evaluation proceeds from left to right. Parentheses may be used to alter the order of evaluation, since by the rules for expressions, any expression enclosed within parentheses will be evaluated first.

If there are two or more relational expressions connected by *and* and *or* operators to form a larger expression, the relational expressions are evaluated from left to right until the truth or falseness of the overall expression can be established and then evaluation of the expression may stop and the appropriate logical value returned.

The relational operators *is* and *is not* are equivalent to the equals (=) and not-equals (/=) relational operators, respectively. This makes comparisons involving names and symbols read more like natural English. For example, the relational expression PUMP1 = ON may alternately be written (and spoken) as PUMP1 is ON.

2.4.4.2. Object Names in Expressions

When an object name appears within an expression, the object to which the name is assigned is substituted into the expression during execution. For example, if SUMX is a name that has been assigned to the real number object 10.0, then the expression 2.0*SUMX will evaluate as 2.0*10.0 and return the value 20.0.

An object name in an expression can be preceded by the reserved word *the*. For example, the command MOVE the MANIPULATOR ARM To the LEFT, where the word *the* appears before both the object of the command and the symbol LEFT (as discussed previously in the definition of symbolic literals in Section B-4.2). If the word *the* appears before an object name in an expression it is ignored and is not considered part of the name. Thus the sample command above has the same meaning regardless of whether one, both or neither occurrences of the word *the* appears.

Object names and symbolic literals overlap in syntactic form — they can both be names — and they may both appear as primaries within expressions. For example, in the relational expression MODE = TRANSPONDER1 the two operands MODE and TRANSPONDER1 can be either object names or symbols. This kind of ambiguity is common in programming languages. A CIL interpreter or compiler can determine during semantic analysis whether a name that appears as a primary in an expression is an object name or a symbolic literal by applying the following test: when a name is encountered as a primary, the list of object names visible at that point is searched; if the primary matches a known object name, then the primary is assumed to be that object name; otherwise, if the primary does not match any visible object name, then the primary is classified as a symbolic literal.

As a result of the rule above, every name appearing in an expression will be classified as either an object name or a symbolic literal. This can create a problem if an undeclared object name appears in an expression (for example, due to mistyping a known name or using a name prior to its declaration). In such cases the name will be incorrectly classified as a symbol. This problem is mitigated by the fact that there are very few operations that can be performed on symbols, and so an undeclared object name will usually be caught when the name appears in some context in which a symbol can't be used (for example, in an addition operation). Thus most, but not all, cases of undeclared object names will be detected during semantic analysis.

2.4.4.3. Lists in Expressions

Lists may appear within expressions but they must always be enclosed within parentheses so that they can be unambiguously identified. Optionally, a list in an expression may be preceded by a name that specifes the class of the list. This allows list subclasses to be manipulated properly within expressions. As an example, two classes VECTOR and MATRIX could be derived from class LIST and defined so that a vector can be multipled by a matrix. Then the expression

VECTOR (1., 0., 0.) * TRANSFORMATION MATRIX

specifies that a unit vector is to be multiplied by a matrix. Without the name VECTOR to indicate the class, the first operand would be interpreted as an object of class LIST and the operation would fail because multiplication is not allowed for general lists. If a list in an expression is not preceded by a name, it is assigned to class LIST.

An attribute list within an expression is converted to an object of class LIST following the rules stated in Section B-4.3. During execution, the current value for each of the specified attributes is substituted into the list. The resultant list may then be operated upon in the expression. For example, the attribute list

(TEMP, PRESSURE of TANK1, TANK2)

will be converted by the rules of Section 3.4.3.4 into a list of the following form:

(((TEMP of TANK1) , (PRESSURE of TANK1)) ,
((TEMP of TANK2) , (PRESSURE of TANK2)))

During execution, the current values for the attributes are substituted, producing a list like the following:

((15.44 DEGC , 45.23 PSI) , (14.79 DEGC , 49.18 PSI))

When an attribute of an explicit list object is referred to in an expression, the list must be enclosed within parentheses (with or without the class name specified). For example, to get the count of members in an explicit list, an expression of the following sort is used:

LENGTH of (“ABCDEFG”, “HIJKLMNOP”)

This will return the value 2, since there are two members in the list. By comparison, the expression (LENGTH of “ABCDEFG”, “HIJKLMNOP”) will return the list (7, 9) as its value, which is the length of the two text strings within the list. Another way to state this rule is that the *of* operator can’t “see” through parentheses.

2.4.4.4. Examples of Expressions

Examples of Simple Expressions with Numeric Objects as Operands

In the following, the name INDEX is assigned to an object of class INTEGER and the names RADIUS, X SUM and Y SUM are assigned to objects of class REAL.

-37

2*(INDEX+6) - 3

3.14159*RADIUS**2

X SUM ** 2 - 2. * X SUM * Y SUM + Y SUM ** 2

Examples of Simple Expressions with Measurement Objects as Operands

In the following, the name AOS is assigned to an object of class DT. The names VEHICLE RANGE and ELAPSED TIME are assigned to measurement objects, the former an object of class LENGTH and the latter an object of class TIME INTERVAL.

- 1 MIN 12.5 SECS

AOS - 10 MINS

2 * VEHICLE RANGE + 3000 FEET

VEHICLE RANGE / ELAPSED TIME

In the last example, the result will be an object belonging to class VELOCITY.

Examples of Simple Expressions with Strings

These examples demonstrate the concatenation operator (&) for strings.

"To be " & " or not to be"

“usr/” & FILE NAME & “.schedule”

The first example above produces the text string "To be or not to be".

Examples of Relational Expressions

The examples below use some of the variables found in the previous examples of simple expressions. Additionally, the names PRIMARY PUMP and BACKUP PUMP are assigned to objects of class PUMP. The results of these expressions belong to the enumeration class LOGICAL and they have either the value TRUE or the value FALSE.

X SUM /= 0.0

DISTANCE < 1 MILE

ELAPSED TIME >= 10 MIN 30 SECS

FILE NAME = “schedule12”

PRIMARY PUMP is ON

(SPEED of PRIMARY PUMP) < 1000 RPM

(PRIMARY PUMP, BACKUP PUMP) = (ON, STANDBY)

In the last example a list is compared for equality with another list. The result is TRUE if each of the members of the first list is equal to the corresponding member of the second list. See the definition of class LIST in Section C-2.15 for further information.

<u>Examples of Membership Tests</u>

In the following, SAFE LIMITS is a name assigned to an object of class RANGE. The results of these expressions are objects of class LOGICAL.

INDEX is within 1..10

(SPEED of PRIMARY PUMP) is within 1000 RPM .. 1500 RPM

(SPEED of PRIMARY PUMP) is not within SAFE LIMITS

<u>Examples of Logical Expressions</u>

Logical expressions can use the logical connectors *and* and *or* to connect relational expressions together. The result of each of these expressions is an object of class LOGICAL.

PRIMARY PUMP is ON and (SPEED of PRIMARY PUMP) < 1000 RPM

PRIMARY PUMP is ON and (SPEED of PRIMARY PUMP) is within SAFE LIMITS

FILE NAME = “schedule12” or FILE NAME = “schedule15”

2.4.4.5. Errors During Expression Evaluation

If an error is encountered during the execution of an expression, an exception is raised. Execution of the statement that includes the expression ceases when the exception is raised and no resultant value is returned. If an exception handler for the exception has been provided within a procedure, the exception handler is initiated. See Sections B-5.5 and B-6 for further information on exception handlers.

There are three exceptions used to report errors encountered during the evaluation of expressions:

- NUMERIC ERROR, which indicates a mathematical error like dividing by zero;
- CONSTRAINT ERROR, which indicates the violation of some restriction or limitation on the values of operands;

- STORAGE ERROR, which indicates that the storage space needed to complete an operation is not available.

These exceptions are described further in Section C-1.2. For the standard objects listed in Part C of this document, the formal definition of each operator action indicates which of these exceptions can occur and under what circumstances.

2.4.5. ASSIGNMENT OPERATIONS

An assignment operation create a correspondence between a name and an object. Assignment operations appear in many CIL statements, including name declarations, class derivations, assignment statements and commands. This section describes the operation of assignment in general. Details on how assignment is used in a particular kind of statement are found in the following sections.

The format of the assignment operation is:

assignment ::= name **:=** expression

The name to the left of the assignment operator must have been declared previously and it must be visible at the point of the assignment. The object class of the expression's result value, determined during semantic analysis, must be either the same class for which the name was declared or a subclass of the name's declared class.

An incompatibility between the class of a name and the class of object to which it is to be assigned can be determined during semantic analysis with one exception: if the name is associated with an enumeration class, the expression result must be one of the objects listed by the attribute CHOICES for the name's enumeration class. This error can often be determined and reported during semantic analysis; if, however, it is not detected until run-time, the exception CONSTRAINT ERROR is raised.

A name must always be assigned to some object. If a name has not been assigned to any other object, it is assigned to the special object called NOTHING. The object NOTHING is not allowed as an operand in any expression or as the target object of a command; therefore, the appearance of an uninitialized name in an expression or command causes an error that will be detected during semantic analysis or at run time. If this error is detected during run-time, the exception CONSTRAINT ERROR is raised. Assignment operations that assign a name to the special object NOTHING are allowed.

<u>Examples of Assignment Operations</u>

AOS TIME := DT ”1999/12/3 12:30:40. UTC”

TIME SINCE AOS := CURRENT TIME - AOS TIME

CURRENT PUMP SPEED := (SPEED of PRIMARY PUMP)

2.4.5.1. Assignment Lists

Some statements incorporate an *assignment list*, where each item is either an expression or an assignment. Assignment lists are a syntactic construct and are not a list object. The format of an assignment list is:

assignment_list ::= assignment_list_item { **,** assignment_list_item}

assignment_list_item ::= assignment | expression

Assignment lists will be described further in the statements that use them.

2.5. STATEMENTS

Statements are sentences in the CIL language. Statements are built from lexical elements and they utilize the basic syntactic structures discussed in the previous chapter. Statements are entered interactively to a CIL interpreter or they are compiled by a CIL compiler. The CIL has the following kinds of statements:

- Name declarations
- Class derivations
- Assignment statements
- Commands
- Sequential control statements
- Conditional control statements
- Iterative control statements

The name declaration, class derivation, assignment and command statements are allowed within interactive, interpretative CIL sessions. They are therefore called interactive statements. Sequential control, conditional control and iterative control statements are only allowed within procedures.

```
CIL_executable ::=      interactive_statement |
                        procedure

interactive_statement ::=      name_declaration |
                               class_derivation |
                               assignment_statement |
                               command

statement ::=      interactive_statement  |
                   sequential_control_statement   |
                   conditional_control_statement  |
                   iterative_control_statement
```

Many of the sequential, conditional and iterative control statements enclose a sequence of zero, one or more statements:

```
sequence_of_statements ::=  { statement }
```

2.5.1. NAME DECLARATION STATEMENT

The name declaration statement creates object names and optionally binds the names to objects. The format of the name declaration statement is:

name_declaration ::= **declare [constant]** declaration **;**

declaration ::= name **:** *class*_name [**:=** expression]

The first name is the name to be declared. Each name must be unique — it must not duplicate a name that is already visible at the point of the declaration. The class name identifies the class of object with which the declared name is to be associated. It must be the name an object class that is visible at the point of the declaration. A name may only be assigned to objects that belong to the class specified in its declaration, or to a subclass of that class.

If the optional expression appears in a declaration, the expression result must belong to the object class named in the declaration or to a subclass of that class. The expression is evaluated when the declaration statement is executed and the name is assigned to the object that is the expression result. If no expression is present in the declaration, then the name is assigned to the special object NOTHING.

If the reserved word *constant* is specified when declaring a name, then the name is bound to the object specified by the expression for the lifetime of the name. The assignment operator and expression must be present for a constant declaration. Once it has been declared, a constant name must not appear on the left-hand side of an assignment operation. A constant name may, however, be used like any other object name in an expression or a command. If the word *constant* is not specified when a name is declared, then the name is a variable name and it is allowed on the left-hand side of an assignment statement. In the case of a variable name, any value assigned to the name is the initial value.

A name declared within a procedure is visible only within that procedure and only from the point of its declaration. The name is deleted when the procedure terminates.

<u>Examples of Name Declaration Statements</u>

Declare SAMPLE NUMBER : INTEGER ;

Declare POSITION, VELOCITY : VECTOR ;

Declare PLOT TITLE : TEXT STRING := "Plot of Battery Temp versus Time" ;

Declare Constant PI : REAL := 3.14159 ;

Declare Constant RADIAN : REAL := 180.0 / PI ;

2.5.2. CLASS DERIVATION STATEMENT

A class derivation statement creates a subclass from an existing class. The capabilities provided through this statement are limited to creating a subclass that varies from its parent only by specifying values for non-constant class attributes. This is particularly useful for creating enumeration classes and list classes. This statement cannot be used to add new actions or attributes to a class; this more extensive tailoring of subclasses is not provided by this version of the specification.

The format of a class derivation statement is:

> class_derivation ::=
> **derive** *subclass*_name **from** *parent_class*_name with_clause **;**
>
> with_clause ::= **with** assignment_list

The subclass inherits the parent class's attributes and actions. The values of inherited non-constant class attributes may be changed using the With clause. The With clause includes one or more assignments. The left-hand side of each assignment must be the name of one of the class attributes inherited from the parent class. That attribute is assigned to the result of the expression on the right-hand side of the assignment, subject to the general rules of assignment stated in Section B-4.5. If more than one attribute is assigned a value in the With clause, the assignments are allowed in any order. An attribute assignment must be unique for the subclass: the name of a particular attribute must not appear on the left-hand side of two or more assignments in a With clause.

A subclass derived within a procedure is visible only within that procedure and only from the point of its derivation. The subclass is deleted when the procedure terminates.

<u>Examples of Class Derivation Statements</u>
The following examples show three different class derivations, along with declarations of objects of the subclass.

```
Derive LINE NUMBER From INTEGER With RANGE := 1 .. 64 ;
Declare CURRENT LINE : LINE NUMBER ;

Derive WEEKDAYS From ENUMERATION With CHOICES:=
                (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY);
Declare CURRENT DAY : WEEKDAY := THURSDAY;

Derive POSITION VECTOR From LIST With  MEMBER CLASS := LENGTH,
                                       CAPACITY := 3 .. 3 ;
Declare SPACECRAFT POSITION : POSITION VECTOR ;
```

2.5.3. ASSIGNMENT STATEMENT

The assignment statement assigns names to objects. The format of the assignment statement is:

> assignment_statement ::= **let** assignment **;**

The general rules for assignment operations given in Section B-4.5 apply to the assignment statement. Additionally, the name on the left-hand side must have been declared to be a variable name.

During execution of an assignment statement, the name on the left-hand side is assigned to the object on the right-hand side. Thereafter, if the name is used in an expression or a command, the expression or command acts upon the object to which the name is assigned. The assignment between a name and an object continues until the name is assigned to another object or until the name is deleted upon exit from the scope in which the name was declared. An object is allowed to have more than one name assigned to it. For example, the assignment

    Let ACTIVE PUMP := PURGE PUMP1 ;

will assign the name ACTIVE PUMP to the same object to which PURGE PUMP1 is assigned. The actual purge pump object could thereafter be referred to by either name. For example, after the above assignment the two commands,

    Turn On PURGE PUMP1;
    Turn On ACTIVE PUMP ;

are equivalent since they refer to the same object.

An attribute of an object is not allowed to appear on the left-hand side of an assignment statement. For example, the assignment

    Let SPEED of PUMP1 := 1234 RPM ;

is not allowed. The CIL can only change objects and their attributes by issuing commands. Using the previous example, the command

    Set SPEED of PUMP1 To 1234 RPM ;

might be issued to instruct the pump to change its speed.

Examples of Assignment Statements

Let AREA := 2.0*PI*RADIUS ;
Let AOS := DT “1995/12/19 05:43:12 UTC”
Let TITLE := "Summary of Position Residuals" ;
Let PUMP LIST := (PUMP1, PUMP2, PUMP3) ;
Let TEST VOLTAGE LIMITS := 27.1 VOLTS .. 29.6 VOLTS ;
Let NEW SPEED := (SPEED of PUMP1) ;

2.5.4. COMMANDS

CIL commands are imperative sentences used to manipulate and control objects. The commands that are allowed for an object class are included in the class's formal definition. The definition for each command provides:

- The name of the command — a word or two words that signify the action that the command performs;

- The names of any qualifiers that can appear in the command;

- The name and class of each parameter associated with a qualifier;

- An indicator of which qualifiers are optional and which are mandatory.

With one exception, commands do not return an object as their result. The exception is the Get command discussed below. This document does not specify what constitutes completion of a command. A command may be considered complete as soon as it is sent, or the software executing the command can wait for some acknowledgement of successful transmission or execution. Typically, if there is a need to verify that a command was actually received by and executed by the recipient, status data should be monitored using statements like the Wait statement or If statement to determine this.

The syntax for commands is divided into two major parts: firstly, a basic command that specifies the action to be taken and the objects to which the action is to be applied; and secondly, optional qualifiers that provide additional information about how the command action is to take place. The format for commands is:

command ::= basic_command [command_qualifiers]

2.5.4.1. Basic Commands

A basic command consists of a list of one or more objects preceded by one or two words that specify the action to be performed on the objects. The format for a basic command is:

basic_command ::= action *command*_list

action ::= verb |
verb modifier |
verb qualifier |
stop verb

verb ::= word

The action part of the command specifies what the command is to do. There are four forms for the action:

- A single word, preferably a verb. For example:

  Open

  Move

  Set

- A verb followed by one of the modifier words defined in Section B-3.1. For example:

  Turn On

  Push In

  Whenever a modifier appears as the second word of a command, it is always interpreted as being the second word of the action, as shown above, rather than as the first word of a symbol that is appearing in the command list.

- A verb followed by one of the qualifiers defined below.. For example:

  Insert Into

  Delete From

- A verb preceded by the keyword *stop*. For example:

  Stop Plotting

  Stop Moving

The command list is an explicit list that specifies the objects to be commanded. It may be either an object list or an attribute list. It is not required that all of the objects in the command list belong to the same class; however, the action to be performed by the command must be defined for all of the object classes represented in the command list.

<u>Examples of Basic Commands with Object Lists</u>

Open VALVE1 ;

Turn On ECLSS PURGE PUMP ;

Insert Into MY DATABASE TABLE ;

Stop Moving the SCAN PLATFORM ;

Turn Off PUMP1, PUMP2, PUMP3 ;

Delete “experiment.dat”, “schedule.*” ;

Print SUMX, SUMY, SUMX**2, SUMY**2 ;

<u>Examples of Basic Commands with Attribute Lists</u>

Reset the TEMPERATURE of the COOLING SYSTEM ;

Set SPEED of PUMP1, PUMP2, PUMP3 ;

Print LENGTH of TITLE STRING ;

2.5.4.2. Command Qualifiers

Commands may specify one or more qualifying clauses that provide additional information about the action to be taken. The format of command qualifiers is:

command_qualifiers ::= qualifying_clause {**,** qualifying_clause}

qualifying_clause ::= qualifier assignment_list

qualifier ::= **after** | **at** | **before** | **by** | **every** | **from** |
into | **to** | **until** | **where** | **with** | **within**

The qualifying clauses allowed for any given command are determined by the definition of the command, as specified in the formal definition of an object class. For a sampling of command definitions, see Section C-2.17, wherein the commands that can act upon procedures are described. A command definition indicates which, if any, of the qualifier keywords may be used in the command and associates each qualifier with one or more *parameters.* A parameter is a name for some property that can be specified in the command. For example, in the command

Point CAMERA #1 At TARGET := the MOON ;.

the qualifer is *At* and there is one parameter named TARGET that is associated with this qualifier. In the command the parameter is assigned a symbol value (MOON). In many commands there will be only one parameter associated with a specific qualifier. If there is only a single parameter for a qualifer then the name of the parameter may be omitted:

Point CAMERA #1 At the MOON ;

If there are two or more parameters associated with a qualifier, then a command may specify each parameter in a separate qualifying clause as in the following:

Rotate the POINTING PLATFORM To DIRECTION := the LEFT
To POSITION := 23.75 DEG ;

or the command may include both parameters in the same qualifying clause:

Rotate the POINTING PLATFORM To DIRECTION := the LEFT,
POSITION := 23.75 DEG

If a qualifier has more than one parameter associated with it, and those parameters are declared to belong to different classes of objects, then the parameter name may be omitted, since the values can be assigned to their proper parameters by class. For instance, using the last example above, the following is a valid command:

Rotate the POINTING PLATFORM To the LEFT To 23.75 DEG ;

A parameter may appear within a command only once. Thus continuing with the example above, the command

Rotate the POINTING PLATFORM To 23.75 DEG To 125.4 DEG ;

will result in an error during semantic analysis because the parameter POSITION is specified more than once.

Qualifying clauses may appear in a command in any order and parameters in a qualifying clause may appear in any order. This document does not define or constrain the class of object associated with each qualifier; The following examples are put forth to illustrate the kinds of things each qualifier keyword might specify:

- After — A date and time after which an action is to occur (DT)
  The relative position at which something is to be placed (SYMBOL)
- At — A date and time at which an action is to occur (DT)
  The location where an action is to occur (SYMBOL)
  The priority at which an action is to take place (SYMBOL)
- Before — A date and time before which the action must occur (DT)
  The relative position at which something is to be placed (SYMBOL)
- By — An amount by which something is to change (Measurement subclass)
  A location next to which something is to be placed (SYMBOL)
- Every — An interval of space or time between repetitions (LENGTH, TIME INTERVAL)
- From — The date and time at which something begins (DT)
  A location from which something is taken (SYMBOL)
  An initial value (Measurement subclass)
- Into — A place where something ends up (SYMBOL)
- To — A date and time where something ends (DT)

A location to which something is moved (SYMBOL)
A target value (Measurement subclass)
A destination pathname (PATHNAME)

- Until — A date and time where something ends (DT)
- Where — A condition that must be satisifed (TEXT STRING)
- With — An object that must be included (SYMBOL)
- Within — A period of time within which something must occur (TIME INTERVAL)
  A place in which something is to be placed (SYMBOL)

Examples of Commands With Qualifying Clauses

Set TEMPERATURE of LAB MODULE To 72 DEGF ;
Move MANIPULATOR ARM To NEW POSITION ;
Rotate WRIST By 12.56 DEG ;
Perform LH2 HEATER TEST At LOW PRIORITY With TEST DURATION := 12 MINS,
BACKUP HEATER := OFF ;
Delete From MY DATABASE TABLE Where "ACTIVITY LEVEL < 3.5" ;
Signal TEMP ALARM With TEMPERATURE := (TEMP of LAB MODULE);
Stop Plotting the RADIATOR TEMPERATURE After DT" 12:30 EST" ;

2.5.4.3. The GET Command

The command Get is defined for all objects. It is the only command that returns a value. The command can be used to get the value of an object or of an attribute of an object. The value is returned in the special object RESULT. The name IT is a synonym for the result object. An example of the use of the Get command is:

GET the SPEED of the PURGE PUMP ;
If IT is within SAFE LIMITS . . .

When an object or object attribute appears as an operand within an expression, the Get command is called implicitly to get the value of the object or attribute. Thus the statement

If the SPEED of the PURGE PUMP is within SAFE LIMITS . . .

Is equivalent to the previous example. The Get command, however, can be useful in certain circumstances unique to monitor and control environments. To demonstrate one such situation, imagine two identicial If statements one after another:

If the SPEED of the PURGE PUMP > 1000 RPM Then . . .
If the SPEED of the PURGE PUMP > 1000 RPM Then . . .

it is possible for the logical expression in the first If statement to evaluate to TRUE and the second to FALSE (or vice versa) because the speed of the purge pump might change at any time. On the other hand, a series of statements like the following will ensure that the two statements act consistently:

GET the SPEED of the PURGE PUMP ;
If the RESULT > 1000 RPM Then . . .
If the RESULT > 1000 RPM Then . . .

2.5.5. SEQUENTIAL CONTROL STATEMENTS

The sequential control statements control the flow of execution within CIL procedures. The sequential control statements are:

```
sequential_control_statement ::=      step_statement  |
                                      goto_statement  |
                                      wait_statement  |
                                      return_statement
```

2.5.5.1. Step Statement

The Step statement may be used to subdivide a procedure into smaller groups of statements. The format of the Step statement is:

step_statement ::= **begin** *step*_name **:**
    sequence_of_statements
    [exception_handlers]
  end *step*_name **;**

The step consists of the following parts:

- A name that identifies the step. The name given to a step must be unique — different from the names of all other steps within the procedure in which the step appears. There is no required naming convention for steps nor are steps required to appear in any particular order. For example, having a step named STEP #3 that precedes STEP #1 is permitted.
- A sequence of statements to be executed. Execution of a step begins with the first statement in the step and continues in sequence through the last statement, unless a Goto statement transfers control out of the step or a Return statement terminates the procedure.

- Zero, one or more exception handlers. The syntax and semantics of exception handlers is given in Section B-6.1 below. If a handler for an exception is defined within a step and the exception is raised while within the step, then that handler is called.

Steps are allowed within steps and there is no limit to the depth to which steps may be nested. Steps are also allowed within a Repeat statement, within a Then or Else clause of an If statement or within a When or Else clause of a Choice statement. Steps may also contain Repeat, If and Choice statements.

Examples of Step Statements

```
Begin STEP1:
    Turn On PUMP1 ;
     . . .
    Turn Off PUMP1 ;
End STEP1 ;

Begin STEP2:
   . . .
   Begin STEP2A:
      . . .
   End STEP2A ;
End STEP2 ;
```

2.5.5.2. Goto Statement

The Goto statement provides the means to alter the sequential flow of execution by specifying the point within a procedure at which execution will continue. The format of the Goto statement is:

goto_statement ::= **goto** *step*_name **;**

A Goto statement cam only transfer control to the beginning of a step. The named step must exist within the procedure in which the Goto is issued. The innermost sequence of statements that encloses the step to which control is to be transferred must also contain the Goto statement (although the Goto statement may be embedded in a deeper sequence of statements within the enclosing sequence). The diagram below depicts this scope rule. The arrows indicate transitions that are allowed. Note that the Goto statement within STEP1A cannot access STEP2A because the latter step is within a sequence of statements that does not enclose the Goto statement (namely the sequence of statements that forms the body of STEP2).

The scope rule prohibits the following:

- Transfer into a Repeat statement, an If statement or a Choice statement from outside such a statement. However, the Goto statement may be used to transfer out of these sequences.
- Transfer between the Then clause and the Else clause in a If statement or from one clause of a Choice statement to another.

Example of Goto Statement

In the following example, the Goto statement within STEP1 is used to skip over STEP2 if the transponder is off.

```
Begin STEP1:
   If TRANSPONDER is ON
      Then
          . . .
       Else
         Goto STEP3 ;
   End If ;
End STEP1 ;

Begin STEP2:
   . . .
End STEP2 ;

Begin STEP3:
   . . .
End STEP3 ;
```

2.5.5.3. Wait Statement

The Wait statement allows a procedure to suspend itself, either indefinitely or for a specified length of time. The format of the Wait statement is:

wait_statement ::= **wait** [*timing*_expression] **;**

Execution of a Wait statement causes the procedure that contains the Wait statement to be suspended. If no timing expression is specified, then execution of the procedure resumes only when the procedure receives a Resume command. If an expression is present, it must evaluate to an object of the measurement subclass TIME INTERVAL. Execution resumes after the specified interval of time has elapsed. If the specified time interval is less than or equal to zero, then no wait occurs.

If a Suspend command is received while a procedure is in a timed wait, then the wait state is converted to an unconditional wait, and only a Resume command can cancel the wait and allow execution to begin again.

It is often useful to suspend execution for a specified length of time or until a specified condition is met, whichever occurs first. The If statement, described in Section B-5.6 below, provides this kind of capability.

Example of Wait Statement

In the following example, a heater is turned on prior to performing an experiment and the Wait statement is used to delay the start of the experiment for 90 seconds to give the experiment chamber time to heat up.

```
Turn On HEATER1 ;
Wait 1 MINUTE 30 SECONDS ;
Perform PLANT GROWTH EXPERIMENT ;
```

2.5.5.4. Return Statement

The Return statement terminates execution of a procedure. A variant is provided to terminate all active procedures. The format of the Return statement is:

return_statement ::= **return** [**all**] **;**

A Return statement within a procedure terminates execution of the procedure. If the terminated procedure was called by another procedure, then control is returned to the caller and execution of the caller continues from the statement following the call.

If the keyword *all* is present in a Return statement, then any procedure that receives control from the returning procedure is itself terminated immediately, as if it encountered a Return All statement. This process continues until all active procedures are terminated.

2.5.6. CONDITIONAL CONTROL STATEMENTS

The conditional control statements allow the execution of a sequence of statements to be contingent upon certain conditions. The conditional control statements are:

conditional_control_statement ::= if_statement |
choice_statement

2.5.6.1. If Statement

The If statement executes one of two blocks of statements, based upon a condition. The format of the If statement is:

```
if_statement ::=
                if logical_expression  [within timing_expression]
                    then_clause
                    [else_clause]
                end if ;

then_clause ::=         then
                            sequence_of_statements

else_clause ::=         else
                            sequence_of_statements
```

The If statement consists of the following parts:

- An expression that is evaluated to determine whether or not a condition is met. For example, the expression

  PUMP1 is On and (Speed of PUMP1) > 1000 RPM

  might be used to determine whether or not the specified pump is operating and up to speed. The expression must evaluate to an object that belongs to class LOGICAL (that is, a value of TRUE or FALSE).

- An optional timing expression that specifies an interval of time during which the logical expression will be re-evaluated. This expression must evaluate to an object of the measurement subclass TIME INTERVAL. If a timing expression is not included in the statement, then the logical expression is evaluated once. If a timing expression is given, the logical expression is periodically re-evaluated until the expression evaluates to TRUE or until the specified time interval has elapsed. The manner in which, and the rate at which, the logical expression is to be re-evaluated is not specified here. It is recommended, however, that the logical expression be re-evaluated at least once per second.

- A Then clause that specifies a sequence of statements that are executed if and when the logical expression evaluates to TRUE.

- An optional Else clause that specifies a sequence of statements that are executed if the logical expression remains FALSE.

Examples of If Statement

If PUMP1 is ON Within 10 SECONDS
  Then

```
    Set the SPEED of PUMP1 To 1234 RPM ;
  Else
    Issue PUMP FAILURE ALERT MESSAGE ;
End If;

If PUMP1 is ON
  Then
    ISSUE PUMP FAILURE ALERT MESSAGE ;
End If ;
```

2.5.6.2. Choice Statement

The Choice statement selects for execution one of a number of alternative sequences of statements. The chosen alternative is determined by the value of an eumerated value. The format of the Choice statement is:

choice_statement ::= **choice** expression
    choice_body
  end choice ;

choice_body ::= when_group [else_clause] |
  else_clause

when_group ::= when_clause
  {when_clause}

when_clause:= **when** *static*_list **then**
    sequence_of_statements

The expression appearing after the keyword *choice* is called the determinant, since the result of this expression determines which, if any, of the clauses of the Choice statement will be executed. The determinant expression must evaluate to an enumeration object (an object that belongs to a subclass of class ENUMERATION).

A Choice statement contains zero, one or more When clauses. If a Choice statement has no When clause then it must have an Else clause, and if a Choice statement has at least one When clause, then the Else clause is optional. The static list within a When clause is an explicit list where each member is a static expression. A static expression is an expression with a constant value that can be evaluated during semantic analysis. A static expression meets the following criteria:

- Only literals, constant names, and the names of class attributes can appear as operands in the expression.

- Evaluation of the expression must not generate an exception. For example, X / 0 is not a valid static expression since division by zero would generate an exception.

Each static expression must yield an object that is included as a member of the list associated with the attribute CHOICES for the enumeration object's class (see Section C-2.12 for further details on the list of choices in an enumeration class). A value must not be listed more than once in a When clause, nor is any value allowed in more than one When clause. If a When clause contains the object that corresponds to the determinant's value, then the statements associated with that When clause are executed. If the object corresponding to the determinant's value does not appear within any When clause, then the statements in the Else clause, if any, are executed.

The following example selects from among three commands, based upon the value of an enumeration object named PRIMARY PUMP STATE::

```
Derive STATUS From ENUMERATION
                                   With CHOICES := (ON, OFF, STANDBY) ;
Declare PRIMARY PUMP STATE : STATUS ;
 . . .
Choice PRIMARY PUMP STATE
  When ON           Then    Set Speed of PRIMARY PUMP to 1234 RPM ;
  When OFF          Then    Turn On BACKUP PUMP ;
  When STANDBY      Then    Turn Off BACKUP PUMP ;
End Choice ;
```

The value of the enumeration object PRIMARY PUMP STATE determines which of the three When clauses is chosen. For example, if the value of PRIMARY PUMP STATE is OFF, then the second When clause is selected and the command to turn on the backup pump is executed.

Examples of Choice Statements

```
Choice (STATE of TRANSPONDER)
    When  OFF                 Then    Turn On TRANSPONDER ;
                                      Set TRANSPONDER To MODE1 ;
    When  MODE1, MODE2        Then    Set TRANSPONDER To MODE3 ;
    When  MODE3               Then    Turn Off TRANSPONDER ;
 End Choice ;

Derive  AVAILABLE PUMPS From ENUMERATION
                        With CHOICES := (PUMP1, PUMP2, PUMP4, PUMP6);

Declare ACTIVE PUMP : AVAILABLE PUMPS ;
```

. . .
Choice ACTIVE PUMP
When PUMP1 Then . . .
When PUMP4 Then . . .
Else . . .
End Choice;

The objects that are the choices for an enumeration object do not need to be symbols: they can be any kind of object. The second example above shows how a Choice statement can be used to select a set of statements for execution based upon which pump device is active. The statements in the Else clause will be executed if the value of ACTIVE PUMP is either PUMP2 or PUMP6.

2.5.7. ITERATIVE CONTROL STATEMENTS

Iterative control statements provide for the repetitive execution of a sequence of statements. A basic looping mechanism is available, along with statements that provide greater levels of control over the execution of the loop. The iterative control statements are:

```
iterative_control_statement ::=    repeat_statement    |
                                   while_statement     |
                                   for_statement       |
                                   exit_statement
```

Repeat Statement

The Repeat statement executes a sequence of statements repeatedly. The format of a Repeat statement is:

repeat_statement ::= **repeat**
sequence_of_statements
end repeat ;

Execution of a Repeat statement begins with the first statement in the specified sequence of statements, continues to the last statement in the sequence, and then begins again at the first statement in the sequence, unless and until an Exit statement, a Goto statement, a Return statement or a Terminate command causes an exit from the Repeat statement.

<u>Example of Repeat Statement</u>

In the following example, the procedure named BATTERY STRESS TEST is called repeatedly until the battery voltage falls below its allowed minimum, at which point the loop is exited using a Goto statement.

Repeat

```
   Perform BATTERY STRESS TEST ;
   If BATTERY VOLTAGE > 25.3 VOLTS
        Then
                Goto SHUTDOWN ;
   End If ;
End Repeat ;
Begin SHUTDOWN:
    . . .
End SHUTDOWN ;
```

2.5.7.1. While Statement

The While statement allows a sequence of statements to be executed for as long as a specified condition holds. The format of the While statement is:

while_statement ::=   **while** *logical*_expression
repeat_statement

The controlling logical expression is evaluated at the beginning of each iteration of the repeat loop. If the control expression evaluates to TRUE then the repeat loop is executed. If the control expression evaluates to FALSE, then execution resumes with the statement immediately following the end of the Repeat statement.

Example of While Statement

The following is based on the same situation used for the Repeat statement example. Note that the While statement removes the need for the If statement and the Goto statement within it.

```
While BATTERY VOLTAGE > 25.3 VOLTS
   Repeat
      Perform BATTERY STRESS TEST ;
   End Repeat ;
```

2.5.7.2. For Statement

The For statement performs a specific number of iterations of a Repeat statement. The format of the For statement is:

for_statement ::=   **for** *index*_name := *index*_list
repeat_statement

The name specified on the left-hand side of the assignment operator is called the index name. It must be a previously-declared variable. The list on the right-hand side of the assignment operator must be an explicit list. The index list may be either an object list or an attribute list. If it is an attribute list, it is converted to an object list prior to execution of the statement. With one exception noted below, each of the members of the index list must belong to the same class as the index name or to a subclass of that class.

When the statement is executed, the expression for the first member of the index list is evaluated and the index name is assigned to the resulting object. The Repeat statement is then executed. Upon completion of the Repeat statement, the second expression in the index list, if any, is evaluated, the index name is set to the resultant value, and the Repeat statement executed again. This process is repeated for each member of the index list in sequence.

If an index expression results in an object of class RANGE, and if the index name is declared to be of class INTEGER, then the Repeat statement is executed once for each integer value in the range, with the index name referring to each new integer value in sequence.

<u>Examples of For Statement</u>

```
For PUMP := PUMP1, PUMP2, PUMP3
  Repeat
     If PUMP is ON
       Then
          Issue PUMP ALERT MESSAGE ;
     End If ;
  End Repeat ;
```

```
For N := 1..5, 15..20
   Repeat
     . . .
   End Repeat ;
```

In the first example above, the name PUMP is assigned in succession to PUMP1, PUMP2 and PUMP3. The loop is executed once for each of the three assignments.

In the second example above, the name N is assigned to the integer objects 1, 2, 3, 4, 5, 15, 16, 17, 18, 19 and 20 in sequence.

2.5.7.3. Exit Statement

The Exit statement can be used to exit from a Repeat statement. The format of the Exit statement is:

exit_statement ::= **exit** [**if** *logical*_expression] **;**

If the optional logical expression is not specified, then the Exit statement causes control to be transferred to the statement immediately following the Repeat statement. If a logical expression is present, then the expression is evaluated. The result must belong to class LOGICAL. If the expression result is TRUE then control is transferred to the statement following the Repeat statement. If the expression result is FALSE, then the execution within the Repeat statement continues with the next statement in sequence after the Exit statement.

Example

The following is based on the same example used for the Repeat and While statements.

```
Repeat
   Perform BATTERY STRESS TEST ;
   Exit If BATTERY VOLTAGE < 25.3 VOLTS ;
End Repeat ;
```

2.6. PROCEDURES

Procedures are sequences of CIL statements that are compiled together and later executed to perform some activity. Procedures are objects, and they belong to the standard object class PROCEDURE. As with other objects, procedures can respond to commands. Therefore, no special CIL statement is provided to start a procedure: a command (the Perform command) is provided for that purpose. For more information on class PROCEDURE and its commands, see Section C-2.17.

A procedure has the following format:

procedure ::= **procedure** *procedure*_name [**(** [parameter_list] **)**] **is**
sequence_of_statements
[exception_handlers]
end *procedure*_name **;**

parameter_list ::= parameter { **,** parameter }

parameter ::= qualifier declaration

exception_handlers ::= when_group

A procedure consists of the following components:

- A name that identifies the procedure.

- An optional set of parameter declarations. Parameters are names to be associated with objects passed to the procedure when called. Each parameter is associated with a qualifier. The rules for representing qualifiers and parameters in a procedure call are the same as described in Section B-5.4 for commands (because procedures are invoked with a command).

- A sequence of statements. Execution of a procedure begins with the first statement in the sequence. Execution of a procedure terminates after the last statement in the body of a procedure has been executed, or when a Return statement is executed, or if a Terminate command is received while the procedure is active, or if an exception occurs within the procedure.

- Zero, one or more exception handlers. If an exception is raised during execution of a procedure, a handler for the exception included within the procedure is activated.

All of the object classes and object names defined as part of the environment in which a procedure executes are visible throughout the procedure. A parameter must be unique at the point of its declaration — it must have a name different from any name visible at that point. The declaration of a formal parameter determines the class or classes of objects to which the name can be assigned. An actual parameter assigned in a Perform command must be an object that belongs to the same class as the formal parameter or to one of the formal parameter's subclasses. For example, if the formal parameter of a procedure is declared to be of class PUMP, then passing an object of a pump subclass like VACUUM PUMP as the actual parameter in one call to the procedure, and an object of a second pump subclass like WATER PUMP in another call to the procedure, is allowed. On the other hand, if the formal parameter was declared for class VACUUM PUMP, then a call that attempts to pass a WATER PUMP object as the actual parameter will result in an error during semantic analysis of the Perform command. Parameters may be passed to a procedure but not back from a procedure.

An expression is allowed in a formal parameter declaration to specify a default value for the parameter. If a default value is given and the parameter is not assigned a value in the command, then the formal parameter name is assigned to the default value. If no default value is provided for a parameter, then the parameter is considered to be mandatory. If a mandatory parameter is omitted from a Perform command, then the exception CONSTRAINT ERROR is signalled.

A procedure may call another procedure, which may in turn call another procedure, and so on. If a procedure calls another procedure via a Perform command that contains a Within clause, then the called procedure executes concurrently with the calling procedure, with each procedure in its own environment. If the Within clause is omitted, then the calling procedure is suspended and the called procedure is executed in the same environment. Upon return from the called procedure, execution of the calling procedure continues from the statement following the Perform command, unless the called procedure terminated with a Return All statement, in which case the calling procedure itself terminates as if it had executed a Return All statement. A CIL procedure is allowed to call itself, although there is little need for recursion in the types of applications for which the CIL is best suited.

<u>Example of a Procedure</u>

```
Procedure PURGE PUMP TEST (With PUMP: VACUUM PUMP)  Is
  Begin STEP1
    Turn On PUMP ;
    If PUMP is ON Within 10 SECONDS
      Then
          Set SPEED Of PUMP To 1234 RPM ;
          Wait Until PUMP SPEED > 1000 RPM ;
           . . .
          Turn Off PUMP ;
     Else
          Perform PUMP TEST SHUTDOWN ;
          Return ;
     End If ;
```

```
  End STEP1 ;
  . . .
End PURGE PUMP TEST ;
```

In the example above, the procedure accepts a single vacuum pump object as its parameter. The procedure would be started with a command like

```
   Perform PURGE PUMP TEST With PUMP := PURGE PUMP #2 ;
```

2.6.1. EXCEPTION HANDLERS

An exception handler consists of when clause followed by one or more statements (see the description of the Choice statement in Section B-5.6 for more information on When clauses). Each static expression value in the When clause must be one of the members of the enumeration subclass EXCEPTION. For example:

```
When COMMAND ERROR Then . . .
When STORAGE ERROR Then . . .
```

If an exception handler is present within a procedure for a particular exception, then when the exception is raised the statements associated with the exception handler are executed. If no exception handler is defined for the exception, then execution of the procedure is either terminated or suspended at the point of the exception.

For finer-grained handling of exceptions, exception handlers are also allowed within a step. If a handler for an exception is present within a step, the statements within that handler are executed when the exceptionis raised. If there is no handler for the exception within the step, then the handler, if any, at the procedure level is executed.

<u>Example of Procedure With Exception Handlers</u>

The following example adds exception handlers at both the step and procedure level to the previous example of a procedure.

```
Procedure PURGE PUMP TEST (With PUMP: VACUUM PUMP)  Is
  Begin STEP1
    Turn On PUMP ;
    If PUMP is ON Within 10 SECONDS
      Then
          Set SPEED Of PUMP To 1234 RPM ;
           Wait Until PUMP SPEED > 1000 RPM ;
           . . .
           Turn Off PUMP ;
```

```
    Else
          Perform PUMP TEST SHUTDOWN ;
          Return ;
    End If ;
    -- Exception handler for STEP1
   When COMMAND ERROR  Then
         Print  "Error encountered while setting up pump for test" ;
         GoTo STEP3 ;
 End STEP1 ;
 . . .
  -- Exception handler for procedure
 When CONSTRAINT ERROR, COMMAND ERROR Then
      Perform PUMP TEST SHUTDOWN ; -- Make sure everything is off
      Return ;
End PURGE PUMP TEST ;
```

In the example above, if a COMMAND ERROR exception occurs within STEP1, then the handler associated with the step will be executed. If a CONSTRAINT ERROR occurs within STEP1, then since there is no handler for such exceptions at the step level the procedure-level handler is executed.

3. FORMAL DESCRIPTION OF STANDARD OBJECTS

3.1. OVERVIEW

This part of the Control Interface specification contains requirements, recommendations, guidelines and options regarding the definition and implementation of the standard object classes. The standard object classes, provided with every implementation of the Control Interface Language (CIL), are listed below. The classes with names given in italics represent *virtual classes.* There are no object instances of virtual classes: they exist only to provide a framework for subclasses from which instances can be created.

OBJECT

- *STANDARD OBJECT*
    - INTEGER
    - *MEASUREMENT*
        - REAL
        - TIME INTERVAL
        - Other subclasses for specific measurement categories
    - *STRING*
        - TEXT
        - DT (Date-Time)
        - BIT STRING
            - B (Binary Bit String)
            - O (Octal Bit String)
            - X (Hexadecimal Bit String)
    - SYMBOL
    - *ENUMERATION*
        - LOGICAL
        - EXCEPTION
        - MONTH
        - TIME ZONE
    - RANGE
    - LIST
    - STATEMENT
    - PROCEDURE

Classes are defined hierarchically, and the indentation in the list above shows the hierarchy of the standard objects. Class OBJECT is the antecedent of all other object classes. All standard object classes are derived from the class STANDARD OBJECT which is a virtual class that exists to define operations that are common to all of the standard objects.

The classes that are derived directly from STANDARD OBJECT — INTEGER, MEASUREMENT, STRING, SYMBOL, ENUMERATION, RANGE, LIST, STATEMENT and PROCEDURE — are called the *base classes* because they are basic to the functioning of the CIL. With the exception of class ENUMERATION, the language provides a specific syntactic representation for each of the base classes; namely the literal forms of integers, measurements, strings and symbols discussed in Section B-4.2, the formats for explicit ranges and lists given in Section B-4.3; and the syntax definitions for statements and procedures found in Sections B-5 and B-6, respectively. The set of base classes is closed: no additional subclasses can be added directly to class STANDARD OBJECT.

Three base classes — MEASUREMENT, STRING, and ENUMERATION — have subclasses that are present in every control interface implementation. These subclasses provide part of the semantics of the CIL. For example, the value computed from a relational expression always belongs to class LOGICAL. The class MEASUREMENT has two mandatory subclasses — TIME INTERVAL and REAL — that must be included in each application. Additional measurement classes may be added, depending upon the needs of a specific application. In the interest of compatibility across applications, an augmented set of standard measurement subclasses is listed in Section C-2.4 below. The recommended implementation of these optional measurement classes — in particular the units of measure that should be supported within these classes — will be provided in a later version of this document.

Each of the object class definitions in this section includes:

- A general discussion of the class and the properties of its objects. A class definition is an interface specification: it does not specify the way in which objects of the class are to be implemented. For example, the specification for class INTEGER does not require integer objects to be 32 bits long, or 64 bits, or whatever.

- A description of the attributes provided by the class to specify important characteristics shared by objects of the class. An example is the attribute called RANGE within class INTEGER that indicates the range of values that can be represented.

- A description of the actions that provide the functionality associated with objects of the class. There are two kinds of actions: operators, and commands. For example, class INTEGER includes the operation of addition and specifies that an integer number can only be added to another integer. The class PROCEDURE, on the other hand, does not include an action for addition, so a procedure can't be added to anything. But commands are provided within class PROCEDURE to initiate, suspend and terminate procedures.

The attributes and actions of an object class are determined by inheritance, meaning that an object has all of the attributes and actions defined for it, plus those of its parent class, its parent's parent, and so on. Every object inherits the attributes and actions specified for class OBJECT because it is the antecedent of all other object classes. Every standard object inherits the attributes and actions of class STANDARD OBJECT.

The attributes and actions listed in the following sections are required in all CIL implementations. Attributes and actions other than those that are described in this specification cannot be provided for standard object classes. This restriction is meant to promote portability of procedures by preventing procedures from incorporating special features found in one implementation but not in others. Since subclasses of the standard classes may be created, it is easy to add new behaviors.

3.1.1. ABOUT ATTRIBUTES

The attributes listed in each object class definition are broken out into two categories:

- CLASS ATTRIBUTES — Attributes that have the same value for all objects of the class. Class attributes are often used to specify the unique limitations or restrictions of each implementation. For example, the attribute MAX LENGTH for class TEXT defines the maximum possible length of a text string. The maximum possible string length is the same for all string objects within a given implementation but it may differ from implementation to implementation.

- INSTANCE ATTRIBUTES — Attributes that have values that are specific to each object of a class. For example, the attribute LENGTH gives the actual number of characters in each particular text string object: for the text string "ABCDE", the value of attribute LENGTH is 5, while for the text string "XYZ" the value is 3.

3.1.1.1. Requirements for Attributes

A class attribute may be defined to have a specific value, in which case the value is constant and is propagated to all subclasses that inherit the attribute. Sometimes the class definitions below don’t explicitly provide the value of a class attribute but instead provides directions on how to assign a proper value for a specific implementation. An example is the set of instructions for assigning the value of the attribute RANGE in class MEASUREMENT. If a class attribute is not given a constant value, then an appropriate value must be assigned within each subclass that inherits the attribute. An example is the attribute UNITS in class MEASUREMENT, which must be assigned to a list describing the units of measure that are allowed for a given measurement class.

Attributes return their value whenever they are referenced within an expression or in a GET command. The value of an attribute may be stored in memory and then retrieved, or it may be computed. Since instance attribute values can be computed on demand, they can perform like functions in other computer languages. For example, mathematical manipulations — like the absolute value or the square root of a real number — which are implemented in other languages as functions are implemented in the CIL as instance attributes.

An object class cannot have two attributes with the same name. However, attributes with the same name may be defined for more than one class, and the meaning of a particular attribute may vary from class to class. For example, the attribute LENGTH is defined for class TEXT and for class BIT STRING, but in the former case the length is measured in characters and in the latter case in bits.

If a reference is made within a CIL expression or command to an attribute that does not exist, then an error is reported during semantic analysis. For example, referring to SPEED of "ABC" results in an error since SPEED is not an attribute for objects of class TEXT.

3.1.2. ABOUT ACTIONS

The actions for each object class are divided into two categories, based upon how an action is invoked:

- OPERATOR ACTIONS — These are actions that are specified in the CIL using operators. For example, A+B invokes the addition action to add two numbers. Operators are defined chiefly for standard data objects like numbers and strings. Other objects, like pumps, will typically not define any arithmetic operators, although a pump object class might provide the relational operators for equality and non-equality so that a CIL procedure can test the state of a pump with a relational expression like the following:

  If PUMP1 is OFF Then ...

- COMMAND ACTIONS — These are actions that are specified using CIL commands. An example is:

  INCREASE SPEED of PUMP1 BY 30 RPM ;

3.1.2.1. Requirements for Actions

A class inherits the actions of its antecedent classes. The way in which an action is carried out is not specified and is implementation-dependent. An action that is inherited by a class may be implemented differently than for the class from which the action is inherited (or for any other class). This is called *polymorphism*. For example, the command PRINT defined in class STANDARD OBJECT will be inherited by every standard object class but may well be implemented differently for integers and strings to take into account the unique rules that have to be used to format values of these types.

If a CIL expression attempts to perform an action that is not defined for a particular object class, or attempts to invoke a defined action with incorrect operands or qualifiers, then an error is reported during semantic analysis.

3.1.2.2. Operator Actions

The definition for each class contains a table defining the operator actions provided for the class, if any. An operator action is defined by a *signature:* a template that shows the operation as it would appear within a CIL expression. The operands in a signature have generic names like Integer1 and TimeInterval2. These names indicate the class of each operand and their position within the sequence of operands. For example, DT1+TimeInterval2 denotes the addition of a date-time object (of class DT) and a time interval object. The first operand is the object upon which the action is taken and the class of the first operand determines which class definition will be searched for the action. In the previous example, the first operand is DT1 and thus class DT will be searched for the action. In addition to showing the form of an action as it appears within the CIL, the operator tables also indicate the class of object that results from the operation.

The operator actions are:

- The unary arithmetic operations

Signature	**Description**
- Operand1	Unary Minus
+ Operand1	Unary Plus

- The binary arithmetic operations

Signature	**Description**
Operand1 + Operand2	Add
Operand1 - Operand2	Subtract
Operand1 * Operand2	Multiply
Operand1 / Operand2	Divide
Operand1 ** Operand2	Exponentiate

- The concatenation operation

Signature	**Description**
Operand1 & Operand2	Concatenation

- The relational operations

Signature	**Description**
Operand1 = Operand2	Equal
Operand1 /= Operand2	Not Equal
Operand1 < Operand2	Less Than
Operand1 > Operand2	Greater Than
Operand1 <= Operand2	Less Or Equal
Operand1 >= Operand2	Greater Or Equal

- The membership operations

Signature	Description
Operand1 **is within** Operand2	Within
Operand1 **is not within** Operand2	Not Within

A class is allowed to have more than one definition of an operator action. This is called *operator overloading* , and it occurs when an operation is defined with more than one type of object as the second operand. For example, two multiplication operations are defined for class INTEGER: in one variation the second operand is an integer number and in the other variation the second operand is a measurement object. Only binary operations can be overloaded, since unary operations don't have a second operand.

Each operator action produces an object as its result. The resultant object need not belong to the same class as the action. For example, the expression 3*5.0 invokes an action of class INTEGER because the first operand is of class INTEGER; however, the result of this addition is an object of class REAL. Some operator actions are listed as returning a result of SELF, indicating that the result belongs to the class from which the operator action is invoked (rather than the class for which the action is defined). For example, an implementation may derive a subclass from class INTEGER called COUNTER. The result of adding two objects of this class will be an object of class COUNTER rather than of class INTEGER.

3.1.2.3. Command Actions

A command specifies a list of one or more objects to be acted upon. The command is applied to each of the objects in the list. The objects in a command need not all belong to the same class. If, however, a command is applied to a class of object for which the command is not defined, then an error is reported during semantic analysis.

For each command described below, the name of the command is given along with a description of each qualifier defined for the command. Qualifiers may be mandatory or optional. An optional qualifier provides a default value that is used whenever the qualifier is not specified in a command invocation.

Unlike operators and actions, commands may not be overloaded. There is only one definition provided for a command. However, allowing optional qualifiers in a command has much the same effect as overloading. In addition, qualifiers can be overloaded: the same qualifier may appear more than once in a command, provided that each use of the qualifier has a different class of value associated with it, so that the qualifiers can be distinguished from one another. For example, the command

Move the ARM To the LEFT To 123 DEG ;

is legitimate because the two values for qualifier To can be unambiguously distinguished (one is a symbol or enumeration object, and the other is a measurement of angle).

Polymorphism is allowed: the same command may be implemented differently based upon its class. For example, assume that there is a virtual class called PUMP that defines a command called Turn On. Specific kinds of pumps are modeled as subclasses of the parent PUMP class. The way in which pumps are turned on may differ depending upon the kind of pump, and thus the implementation of the Turn On command will differ in implementation from one pump subclass to another. In the (hopefully rare) event that a command is not appropriate for a particular subclass, it can be implemented as a *No Op*: that is, an implementation that accepts the command but performs no real action..

3.1.2.4. Errors During Actions

Most class definitions specify requirements regarding the run-time behavior of objects by stating rules about what constitutes an error condition and what to do if an error occurs. If an error is detected while executing an action, then an exception is raised. An exception is represented as an enumerated value of class EXCEPTION. The following standard exceptions are used:

- NUMERIC ERROR — This exception is raised if an error occurs during the performance of an arithmetic operation, including division by zero and overflow during addition, subtraction, multiplication, division or exponentiation.

- CONSTRAINT ERROR — This exception is raised when a limitation or restriction defined for an object class would be violated by an action. For example, each implementation defines the maximum number of characters allowed in a text string; trying to store more characters than this in a string will result in CONSTRAINT ERROR being raised. CONSTRAINT ERROR is also raised if the special object NOTHING appears in an action. This would be the case, for example, if an uninitialized variable name appeared within an expression. Attempting arithmetic operations on measurements with incompatible units also results in CONSTRAINT ERROR. For example, the addition operation 1 FOOT + 3 SECONDS will raise CONSTRAINT ERROR.

- STORAGE ERROR — This exception is raised if an action requires additional storage space for objects— either in memory or in secondary storage — and the needed space is not available.

- COMMAND ERROR — Errors detected during the performance of a command are raised with this exception.

3.2. THE STANDARD OBJECTS

3.2.1. OBJECT

Parent Class: None

Class OBJECT is the starting point for deriving all other classes. It stands at the top of the object class hierarchy and its attribute and actions are inherited by all other object classes. Objects of this class do not have any physical existence and there is little need to declare objects of class OBJECT. However, when the CIL evaluates a statement and cannot determine that an object belongs to any other class, it is assigned to class OBJECT. For this reason, class OBJECT is not technically a virtual class.

3.2.1.1. Class Attributes

CLASS : SYMBOL

For each object class this attribute must be set to the name of the class. For class OBJECT, this attribute is set to the symbol OBJECT; for class INTEGER this attribute is set to INTEGER, and so on.

3.2.1.2. Instance Attributes

There are no instance attributes for class OBJECT.

3.2.1.3. **Operator Actions**

Signature	**Description**	**Result**
Object1 **is within** List2	Within	LOGICAL
Object1 **is not within** List2	Not Within	LOGICAL

<u>Note</u>

1) The “is within” operation returns the value TRUE if and only if the object specified as the first operand is a member of the list. The “is not within” operation returns the value TRUE if the object is not a member of the list.

3.2.1.4. Command Actions

GET

The Get command retrieves the value or values of the objects to which it is applied. This command is unique in that it returns a value. The value is always placed into the special object named RESULT. The name IT is a synonym for the result object. The result object takes on the class of its value. Since the Get command is defined on class OBJECT, and since all attributes are objects, then the command can be used to get the value of attributes. When the value of an object or an attribute of an object is needed as an operand within an expression, the Get command is implicitly issued to get the value.

Example of the Get Command

In the following example object PUMP1 is a physical object of a class like PUMP and SAFE LIMITS is an object of class RANGE.

Get SPEED of PUMP1 ;
If IT is within SAFE LIMITS Then . . .

Note

The above example is equivalent to (and more commonly expressed as)

If SPEED of PUMP1 is within SAFE LIMITS Then . . .

In the latter case, the call to the GET command is implicit.

3.2.2. STANDARD OBJECT

Parent Class: OBJECT

Class STANDARD OBJECT is the parent class for all standard objects. It is a virtual class that defines two command operations for potential use in formatting the output of the values of standard objects.

3.2.2.1. Class Attributes

There are no class attributes for class STANDARD OBJECT.

3.2.2.2. Instance Attributes

There are no instance attributes for class STANDARD OBJECT.

3.2.2.3. Operator Actions

There are no operator actions for class OBJECT.

3.2.2.4. Command Actions

PRINT

The Print command converts a standard object's value into a character string and writes out the converted value, typically to a display window or a file. An implementation may add qualifiers (typically the To qualifier) to this command to direct the printed output to a specific location. This command may be implemented as a *No Op* if the implementation does not support printed output. Example:

PRINT SPEED OF PUMP1, PUMP2, PUMP3 ;

SAY

The Say command converts the value of a standard object to an implementation-specific format and directs the output to a speech-synthesizing device. This command is chiefly for applications which use a voice interface rather than a graphical user interface. An implementation may add qualifiers to direct the speech to a specific location or to otherwise determine the nature of the output. The command may be implemented as a *No Op* if the implementation does not support speech output. Example:

SAY CURRENT TIME;

3.2.3. INTEGER

Parent Class: STANDARD OBJECT

Objects of class INTEGER represent both positive and negative integer numbers. Limitations on the integer values that can be represented by this class or a subclass derived from it are specified through the attribute RANGE.

3.2.3.1. Class Attributes

RANGE : RANGE

The range of integer numbers that can be represented, from the largest negative integer to the largest positive integer. For class INTEGER this attribute should be set to a common representation of integer numbers (for example the range of a 32-bit signed twos-complement integer). Subclasses of class INTEGER can then be created for applications that want to create objects with restricted (or larger) integer ranges. For example:

Derive COUNTER From INTEGER With RANGE := 1 .. 100;

3.2.3.2. Instance Attributes

ABS : SELF

This attribute returns the absolute value of the integer object.

REAL : REAL

This attribute returns the value of the integer object converted to a real number (an object of class REAL).

3.2.3.3. Operator Actions

Signature	**Description**	**Result**
- Integer1	Unary Minus	SELF
+ Integer1	Unary Plus	SELF
Integer1 + Integer2	Add	SELF
Integer1 - Integer2	Subtract	SELF
Integer1 * Integer2	Multiply	SELF
Signature	**Description**	**Result**
Integer1 * Measurement2	Multiply	See Note 1
Integer1 / Integer2	Divide	SELF
Integer1 / Measurement2	Divide	See Note 1
Integer1 ** Integer2	Exponentiate	SELF
Integer1 = Integer2	Equal	LOGICAL
Integer1 /= Integer2	Not Equal	LOGICAL
Integer1 < Integer2	Less Than	LOGICAL
Integer1 > Integer2	Greater Than	LOGICAL
Integer1 <= Integer2	Less Or Equal	LOGICAL
Integer1 >= Integer2	Greater Or Equal	LOGICAL
Integer1 **is within** Range2	Within	LOGICAL
Integer1 **is not within** Range2	Not Within	LOGICAL

<u>Notes</u>

1) If the second operand of a multiplication or division operation is a measurement, then the integer operand is converted to a measurement of class REAL and the corresponding operation for class MEASUREMENT is performed.

2) The remainder, if any, from the operation of division with integer operands is discarded.

3) The exponentiation operation is defined for integer exponents only. The exponent must be greater than or equal to zero; otherwise the exception CONSTRAINT ERROR is raised. An exponent of zero results in the value one.

4) For membership operations, a number is within the range if it is greater than or equal to the range's L value and less than or equal to the range's R value.

5) Overflow during any arithmetic operation or dividing by zero causes the exception NUMERIC ERROR to be raised.

Command Actions

There are no command actions for class INTEGER.

3.2.4. MEASUREMENT

Parent Class: STANDARD OBJECT

Measurement objects express values in units like Volts and Miles per Hour. Each category of measurement that is supported is represented by a subclass of class MEASUREMENT. The following measurement subclasses are included in every implementation:

- REAL
- TIME INTERVAL

Additionally, the following measurement classes are optional, and some or all of them may be included in an implementation:

- LENGTH
- AREA
- VOLUME
- WAVE NUMBER
- AMOUNT
- MASS
- DENSITY
- VELOCITY
- ACCELERATION
- FREQUENCY
- FORCE
- PRESSURE
- ENERGY
- POWER
- FORCE MOMENT
- CURRENT
- CHARGE
- VOLTAGE
- MAGNETIC FIELD STRENGTH
- TEMPERATURE
- LUMINOUS INTENSITY
- LUMINANCE
- ANGLE

- ANGULAR VELOCITY
- ANGULAR ACCELERATION
- SOLID ANGLE
- LUMINOUS FLUX
- ILLUMINANCE
- INFORMATION LENGTH
- INFORMATION RATE
- FLOW RATE

A measurement object is classified into the proper subclass by comparing the object's units of measure to the known units for each subclass. A measurement expressed in feet or meters, for example, will be classified as an object belonging to the measurement subclass LENGTH. For each subclass one or more units of measurement may be defined. For example, length may be measured in feet or meters or both. Several scales of measurement may also be provided. For example, lengths may allows measurements specified in miles, feet and inches. Each measurement subclass sets the attribute UNITS to a list defining the units of measure supported for the subclass. Each subclass also identifies one unit of measurement to serve as the standard units in which all measurements of the subclass will be represented internally. Having all measurements of a subclass expressed in common units simplifies arithmetic and relational operations since no change of scale is required when an operation takes place.

The operator actions defined below apply to all measurement classes. Specifying the operations that are appropriate for each subclass of measure-ment, and indicating the result of each operation is a relatively complex task. For example, adding a length to a length produces a length; but multiplying a length by a length produces an area. There is a set of transition rules that determines what happens to measurements when they are multiplied or divided or subjected to exponentiation. There are several ways of implementing these transition rules, but the implementation is not discussed in this specification.

3.2.4.1. Class Attributes

RANGE : RANGE

This attribute should be set to the range of real values that can be represented, from the largest negative number to the largest positive number. This attribute should be set to a common representation of real numbers (for example the range of a 32-bit IEEE floating point number).

UNITS : LIST

A list of the units of measure associated with the class. The list is empty for class MEASUREMENT and class REAL. For every other measurement class, there is one or more items in this list. Each member of this list is itself a list of the form (Names, A, B), where Names is a symbol or a list of symbols giving the name of one or more units of measure and A and B are real numbers such that Y:=A+B*X, where X is a measurement value expressed in the named units and Y is the measurement value converted to the standard units for the subclass. For an example, see the definition of this attribute for class TIME INTERVAL below.

3.2.4.2. Instance Attributes

ABS : SELF

This attribute returns the absolute value of the measurement object.

3.2.4.3. Operator Actions

Signature	**Description**	**Result**
- Measurement1	Minus	SELF
+ Measurement1	Plus	SELF
Measurement1 + Measurement2	Add	SELF
Measurement1 - Measurement2	Subtract	SELF
Measurement1 * Measurement2	Multiply	See Note 4
Measurement1 * Integer2	Multiply	SELF
Measurement1 / Measurement2	Divide	See Note 4
Measurement1 / Integer2	Divide	SELF
Measurement1 ** Integer2	Exponentiate	See Note 5
Measurement1 = Measurement2	Equal	LOGICAL
Measurement1 /= Measurement2	Not Equal	LOGICAL
Measurement1 < Measurement2	Less Than	LOGICAL
Measurement1 > Measurement2	Greater Than	LOGICAL
Measurement1 <= Measurement2	Less Or Equal	LOGICAL
Measurement1 >= Measurement2	Greater Or Equal	LOGICAL
Measurement1 **is within** Range2	Within	LOGICAL
Measurement1 **is not within** Range2	Not Within	LOGICAL

Notes

1) The result of the unary arithmetic operations belong to the same measurement subclass as the operand. For example, negating a voltage results in a voltage.

2) Measurements can only be added and subtracted if they belong to the same class of measurement. The result belongs to the same measurement class. For example, a length can only be added to (or subtracted from) another length, and the result is a length. If the operands are not of the same class, the exception CONSTRAINT ERROR is raised.

3) When a measurement is multiplied or divided by an integer, the integer is converted to a measurement of class REAL prior to the operation. The result belongs to the same class as the measurement operand.

4) When a measurement is multiplied or divided by another measurement, the result belongs to a measurement class that is determined by the set of transition rules for measurements. If the operation would not result in a valid transition, the exception CONSTRAINT ERROR is raised.

5) The exponentiation operation is defined for integer exponents only. The exponent must be greater than or equal to zero; otherwise the exception CONSTRAINT ERROR is raised. An exponent of zero results in the real value one. The result belongs to a measurement class that is determined by the set of transition rules for measurements. If the operation would not result in a valid transition, the exception CONSTRAINT ERROR is raised.

6) The relational operations for measurements require that both operands belong to the same measurement class. If they are not, the exception CONSTRAINT ERROR is raised.

Command Actions

There are no command actions for class MEASUREMENT.

3.2.5. REAL

Parent Class: MEASUREMENT

This class is used to represent real numbers. They are treated, in essence, as a measurement of distance along the real number line.

3.2.5.1. Class Attributes

UNITS : LIST := ()

There are no units associated with class REAL.

3.2.5.2. Instance Attributes

INTEGER : INTEGER

This attribute returns the value of the real object converted to an integer number (object of class INTEGER).

SQRT : REAL

This attribute returns the square root of a real object.

Notes

1) If some of the optional standard measurement subclasses are supported by an implementation, then some additional attributes may be inserted into this class. In particular, if the measurement class ANGLE is supported, then instance attributes ASIN, ACOS, ATAN should be added to class REAL.

3.2.5.3. Operator Actions

There are no further operator actions for class REAL.

3.2.5.4. Command Actions

There are no command actions for class REAL.

3.2.6. TIME INTERVAL

Parent Class: MEASUREMENT

TIME INTERVAL objects are used to express intervals of time (as opposed to calendar dates and times, which are represented using the DT class).

3.2.6.1. Class Attributes

UNITS : LIST := (((DAY, DAYS) , 0.0, 86400.0) ,
((HOUR, HOURS, HR, HRS), 0.0, 3600.0) ,
((MINUTE, MINUTES, MIN, MINS), 0.0, 60.0) ,
((SECOND, SECONDS, SEC, SECS), 0.0, 1.0))

The standard units for time intervals as given above is seconds.

3.2.6.2. Instance Attributes

There are no instance attributes for class TIME INTERVAL.

3.2.6.3. Operator Actions

Signature	Description	Result
TimeInteval1 + DT2	Add	DT

Notes

1) Time intervals may be added to DT (date-time) objects, yielding a date-time.

3.2.6.4. Command Actions

There are no command actions for class TIME INTERVAL.

3.2.7. STRING

Parent Class: STANDARD OBJECT

Objects of class STRING are a generalization of strings that encompasses text strings and bit strings, as well as other kinds of objects (like date-time strings). This is a virtual class and there are no objects of class string: all string objects belong to a subclass of this class.

3.2.7.1. Class Attributes

There are no class attributes for this class.

3.2.7.2. Instance Attributes

There are no instance attributes for this class.

3.2.7.3. Operator Actions

There are no operator actions for this class.

3.2.7.4. Command Actions

There are no command actions for this class.

3.2.8. TEXT

Parent Class: STRING

Objects of class TEXT represent strings of characters. The case of alphabetic characters within text strings is preserved. The characters in a text string are numbered from one, starting with the first (leftmost) character in the string. An empty (null) string — a text string with no characters — is allowed. See Section B-4.2 for a further discussion of the semantics of text strings.

3.2.8.1. Class Attributes

MAX LENGTH : INTEGER

The maximum number of characters allowed in a text string.

3.2.8.2. Instance Attributes

LENGTH : INTEGER

The number of characters in a text string object, in the range 0 .. MAX LENGTH.

3.2.8.3. Operator Actions

Signature	**Description**	**Result**
Text1 & Text2	Concatenate	TEXT
Text1 = Text2	Equal	LOGICAL
Text1 /= Text2	Not Equal	LOGICAL

<u>Notes</u>

1) When concatenation of text strings is performed, the characters in the first operand string become the first (leftmost) characters in the resultant string and the characters in the second operand string become the last (rightmost) characters of the result. The result has a length equal to the sum of the lengths of the two strings. If the resultant text string would be longer than the maximum length allowed, then exception CONSTRAINT ERROR is raised.

2) The case of alphabetic characters within text strings is significant in comparisons of strings. For example, the string "ABC" does not match the string "abc" in a relational comparison.

3.2.8.4. Command Actions

There are no command actions for class TEXT.

3.2.9. BIT STRING AND ITS SUBCLASSES B, O AND X

Parent Class: STRING

Objects of class BIT STRING represent a string of bits. They may be used to represent non-numeric data like flags, and so on. See Section B-4.2 for further information on the semantics of bit strings.

The bits within a bit string are numbered from one, starting with the leftmost bit (the most significant bit) and proceeding to the least significant bit. There are three subclasses of bit strings that are used to represent bit strings as literals or during input and output, but the internal representation is the same in all cases:

B — Binary

O — Octal

X — Hexadecimal

3.2.9.1. Class Attributes

MAX LENGTH : INTEGER

The maximum number of bits allowed in a bit string.

3.2.9.2. Instance Attributes

LENGTH : INTEGER

The actual number of bits in a bit string object, in the range 1 .. MAX LENGTH.

3.2.9.3. Operator Actions

Signature	Description	Result
BitString1 & BitString2	Concatenate	BIT STRING
BitString1 = BitString2	Equal	LOGICAL
BitString1 /= BitString2	Not Equal	LOGICAL
not BitString1	Not	BIT STRING
BitString1 **and** BitString2	And	BIT STRING
BitString1 **or** BitString2	Or	BIT STRING

Notes

1) When concatenation of bit strings is performed the bits in the first operand string become the most significant bits in the result and the bits in the second operand string become the least significant bits of the result. The result has a length equal to the sum of the lengths of the operand bit strings. If the resultant bit string would be longer than the maximum length allowed, then the exception CONSTRAINT ERROR is raised.

2) The relational operations compare the operand strings bit by bit and if all the corresponding bits within both operands are equal, then the two strings are equal; otherwise the bit strings are not equal. If one bit string is shorter than the other, then the comparison acts as if the shorter string is padded with zeroes in the most significant bits out to the length of the longer string.

3) The **not** operation reverses the binary state of every bit of the operand: zero to one; one to zero.

4) The **and** and **or** operations act upon the corresponding bits of each of the two bit string operands. If one bit string is shorter than the other, then the comparison acts as if the shorter string is padded with zeroes in the most significant bits out to the length of the longer string. The result is a bit string with the same length as the longer of the two operands.

3.2.9.4. Command Actions

There are no command actions for class BIT STRING.

3.2.10. DT

Parent Class: STRING

Objects of class DT represent the combination of a date in the Gregorian calendar and a time of day. Every implementation must be able to handle (at a minimum) all dates within the twentieth and twenty-first centuries, including leap-year calculations as necessary. For further information on the semantics of date-time strings, see Section B-4.2.

Date-time objects are specified relative to a time zone and automatic conversion from one time zone to another is performed when date-times from different zones are acted upon. For example, if a relational comparison is made between 12:00 UTC and 17:00 EST on the same day, then an appropriate adjustment for the difference in time zones will be made before making the comparison and the two times will be recognized as being equal.

The time zones supported by an implementation are determined by the enumeration class TIME ZONE defined in Section C-2.12 below Each implementation will support at least the following time zones and recognize the associated abbreviations as time zone names:

Time Zone Name	Abbreviations	Hours from Greenwich
Coordinated Universal Time	Z, GMT, UTC	0
Eastern Daylight Time	EDT	-4
Eastern Standard Time	EST	-5
Central Daylight Time	CDT	-5
Central Standard Time	CST	-6
Mountain Daylight Time	MDT	-6
Mountain Standard Time	MST	-7
Pacific Daylight Time	PDT	-7
Pacific Standard Time	PST	-8

If no time zone is explicitly included in a date-time value, then it is assumed to be Coordinated Universal Time. Other time zones may be supported within an implementation as needed. Since there are time zones in the world that differ from Greenwich by some non-integral number of hours, nothing in an implementation of date-time objects should preclude handling time zones that differ by a fractional part of an hour.

3.2.10.1.Class Attributes

TICK : TIME INTERVAL

The smallest time difference that can be represented in a clock time. For example, on a computer that has an internal clock with microsecond resolution, this would be set to 1.0E-6 SECONDS.

3.2.10.2.Instance Attributes

The following instance attributes return components of a date-time object:

YEAR: INTEGER
MONTH: INTEGER
MONTH NAME: SYMBOL
DAY: INTEGER
DOY: INTEGER
TIME: TIME INTERVAL
ZONE: SYMBOL

Note

The YEAR, MONTH, DAY and DOY attributes respectively represent the year (all four digits), month, day-of-month and day-of-year. The attribute MONTH NAME is the full name of the month (for example, NOVEMBER). The TIME attribute is the elapsed time since midnight of the day, relative to the time zone associated with the object. The ZONE attribute is the name of the time zone (for example, CDT) or the default zone name (UTC) if a zone was not explicitly specified as part of the object.

3.2.10.3.Operator Actions

Signature	Description	Result
DT1 + TimeInterval2	Add	DT
DT1 - TimeInterval2	Subtract	DT
DT1 - DT2	Subtract	TIME INTERVAL
DT1 = DT2	Equal	LOGICAL
DT1 /= DT2	Not Equal	LOGICAL
DT1 < DT2	Less Than	LOGICAL
DT1 > DT2	Greater Than	LOGICAL
DT1 <= DT2	Less Or Equal	LOGICAL
DT1 >= DT2	Greater Or Equal	LOGICAL

Notes

1) The operations of addition and subtraction are defined so that an interval of time (an object of the measurement subclass TIME INTERVAL) can be added to or subtracted from a date-time. One date-time may also be subtracted from another, yielding the interval of time between them.

2) If an arithmetic operation would result in a date-time value outside the range of dates that can be represented in a particular implementation, then the exception CONSTRAINT ERROR is raised.

3.2.10.4.Command Actions

There are no command actions for class DT.

3.2.11. SYMBOL

Parent Class: STANDARD OBJECT

Objects of class symbol represent values that are represented in words, like the months of the year, the status of a machine, and so on. Examples are JANUARY and HIGH GAIN MODE.

3.2.11.1.Class Attributes

There are no required class attributes for class SYMBOL.

3.2.11.2.Instance Attributes

There are no required instance attributes for class SYMBOL.

3.2.11.3.Operator Actions

Signature	Description	Result
Symbol1 = Symbol2	Equal	LOGICAL
Symbol1 /= Symbol2	Not Equal	LOGICAL

<u>Note</u>

1) A symbol object is equal to another symbol object if and only if they are same symbol; that is, the same string of characters, after any embedded space or horizontal tab characters have been removed and after all alphabetic characters have been converted to upper case. For example, the symbols HIGH GAIN, HIGHGAIN and HighGain are equal, because spaces between the words of a symbol are not significant, nor is the case of alphabetic characters.

3.2.11.4.Command Actions

There are no command actions for class SYMBOL.

3.2.12. ENUMERATION

Parent Class: STANDARD OBJECT

Enumeration objects consist of an object drawn from the set of objects that are members of a list defined by attribute CHOICES. For class ENUMERATION, the attribute CHOICES is initialized to an empty list. This means that class ENUMERATION is not useful in and of itself for creating enumeration objects; however, subclasses of

ENUMERATION can set CHOICES to a meaningful list of values. Useful objects may then be created from these classes. For example, an enumeration class SWITCH STATE might be derived with CHOICES := ON, OFF, STANDBY. An object belonging to class SWITCH STATE could then have any of these three values, depending presumably upon the actual state of a switch object.

Enumeration objects are not restricted to symbols: any kind of object may appear in a list of choices. For example, an enumeration class called AVAILABLE PUMPS might have choices PUMP1, PUMP2, PUMP4. An object of this class could have the value PUMP2 but not the value PUMP3 which is not in the list of choices. The value of an enumeration object takes on the class of its value. In the SWITCH STATE example given above, the value would be a symbol, since all three choices are symbols. In the AVAILABLE PUMPS example, the class of the enumeration value would belong to class PUMP.

There is are four standard subclasses of class ENUMERATION:

LOGICAL

EXCEPTION

MONTH NAME

TIME ZONE

Class LOGICAL is used in relational expressions and logical comparisons. It has some unique operations and so is described in full in the following section. The other three classes are typical enumeration classes, differing only in the set of choices they offer; so they are simply summarized here rather than in separate sections.

3.2.12.1.Class Attributes

CHOICES : LIST

A list containing the set of objects that an enumeration object may have has its value. This attribute is set to an empty list for class ENUMERATION. Each item of the choice list is an object or a list of objects. If a choice list item is itself a list, then the choices in the inner list are assumed to be synonymous. For example, in the definition of enumeration class MONTH NAME in the table below, each month is represented with its full spelling, a three-letter abbreviation, and an integer number in the range 1 .. 12.

The choices for the standard enumeration classes are given in the following table.

Enumeration Class	**Choices**
LOGICAL	TRUE, FALSE
EXCEPTION	NUMERIC ERROR, CONSTRAINT ERROR, STORAGE ERROR, COMMAND ERROR

MONTH NAME	(JANUARY, JAN, 1), (FEBRUARY, FEB, 2), (MARCH, MAR, 3), (APRIL, APR, 4), (MAY, 5), (JUNE, JUN, 6), (JULY, JUL, 7), (AUGUST, AUG, 8), (SEPTEMBER, SEP, 9), (OCTOBER, OCT, 10), (NOVEMBER, NOV, 11), (DECEMBER, DEC, 12)
TIME ZONE	(GMT, UTC, Z, 0), (EDT, -4), (EST, CDT, -5), (CST, MDT, -6), (MST, PDT, -7), (PST, -8)

3.2.12.2.Instance Attributes

SYNONYMS : LIST
This attribute provides a list of the synonyms of the current value, if any. If there are none, the list is empty.

3.2.12.3.Operator Actions

An enumeration object takes on the class of its value and the operations that are defined for that class can be applied to the enumeration object. In the case of the SWITCH STATE example above: since the value is a SYMBOL, the operations defined for symbol — equals and not equals — can be applied.

3.2.12.4.Command Actions

There are no command actions for class ENUMERATION.

3.2.13. LOGICAL

Parent Class: ENUMERATION

The enumeration subclass LOGICAL indicates the result of logical operations in CIL expressions. The choices for class LOGICAL are TRUE — corresponding to logical truth — and FALSE. All relational comparisons and membership comparisons in the CIL result in an object of class LOGICAL. The logical **not** operator and the logical connectors **and** and **or** are defined for class LOGICAL.

3.2.13.1.Class Attributes

CHOICES : LIST := FALSE, TRUE

3.2.13.2.Instance Attributes

There are no instance attributes for class LOGICAL.

3.2.13.3.Operator Actions

Signature	**Description**	**Result**
not Logical1	Not	LOGICAL
Logical1 **and** Logical2	And	LOGICAL
Logical1 **or** Logical2	Or	LOGICAL

<u>Notes</u>

1) The **not** operation returns the value TRUE if the operand is FALSE, or FALSE if the operand is TRUE.

2) The **and** operation returns TRUE if both of the operands are TRUE; otherwise it returns FALSE.

3) The **or** operation returns TRUE if either or both of the operands are TRUE; otherwise it returns FALSE.

3.2.13.4.Command Actions

There are no command actions for class LOGICAL.

3.2.14. RANGE

Parent Class: STANDARD OBJECT

Range objects consist of two numbers that indicate the lower and upper bounds of a range of values. Ranges appear chiefly in the membership operations within and not within that are defined for the number object classes.

The lower and upper bounds of a range are referred to as the L value and R value of the range, respectively. The L value must always be less than or equal to the R value. The L and R values for a range must both belong to the same number class: either class INTEGER or a specific measurement class. For example, the range 12.3 DEGC .. 45.2 DEGC is valid since both the L and R values represent temperatures, but the range 12.3 DEGC .. 12.5 FEET is not allowed since it mixes measurement classes. A range that specifies different units within the same subclass is allowed. For example, 12.3 DEGC..83.4 DEGF is allowed since both values are a measure of temperature.

3.2.14.1.Class Attributes

There are no class attributes for class RANGE.

3.2.14.2.Instance Attributes

LOWER : INTEGER or MEASUREMENT

HIGHER : INTEGER or MEASUREMENT

Note

1) The attributes LOWER and UPPER return the lower (L) value and the upper (R) value of the range, respectively. The class of the result belong to class INTEGER or a subclass of MEASUREMENT, depending upon the class of the two number objects contained in the range object.

3.2.14.3.Operator Actions

There are no operator actions for class RANGE.

3.2.14.4.Command Actions

There are no command actions for class RANGE.

3.2.15.LIST

Parent Class: STANDARD OBJECT

Lists are objects that hold other objects. The objects in a list are called the list's members. Objects of class LIST can hold any number of members of any object class. Subclasses of class LIST may be derived to constrain either the number of members allowed in a list, or the class of the list members, or both. This allows data structures like arrays to be created. Lists are allowed to have list objects for members; this feature can be used to develop data structures that act like multi-dimension arrays.

Any limitations on the number of list members are established by setting the attribute CAPACITY. Limitations on the class of members are established by setting the attribute MEMBER CLASS. A list is restricted to having members that belong to the class specified by the MEMBER CLASS attribute or one of its subclasses. For example, a list subclass named UNIT VECTOR could be created using CAPACITY:= 3..3 and MEMBER CLASS:= REAL.

The members of a list are numbered, starting with the number one for the first list member. If the Nth list member is itself a list, then the first member of N is identified as (N,1), the second member as (N,2), and so on.

3.2.15.1.CLASS Attributes

CAPACITY : RANGE

The number of list members allowed. The L value establishes the fewest number of members allowed and the R value the most number of members allowed. The values for this attribute for class LIST are 0 .. M, where M is the maximum number of members any list may have in an implementation.

MEMBER CLASS : SYMBOL

This attribute specifies the class of object that a list can have for its members. Each member of the list must belong to this class or one of its subclasses. The default value is OBJECT, which allows a list to contain any class of object.

3.2.15.2.Instance Attributes

LENGTH : INTEGER

The number of members in a list. The value must be within the range specified by the attribute CAPACITY. A list with no members is called an empty list.

3.2.15.3.Operator Actions

Signature	**Description**	**Result**
List1 = List2	Equal	LOGICAL
List1 /= List2	Not Equal	LOGICAL

<u>Note</u>

1) The relational operations compare the members on each of the two operand lists. The first members of each list are compared to each other, as are the second members, and so on. The lists must have the same number of members, or the list is not equal. The two members are compared using the relational operation defined for the object class to which the member of the first operand list belongs. If the relational operation is not defined for the two list member's classes, then an error is reported during semantic analysis. The result of the Equal operation is TRUE if and only if all members of the lists are equal to each other. The result of the Not Equal operation is TRUE if any or all of the list members are not equal to each other.

3.2.15.4.Command Actions

There are no command actions for class LIST.

3.2.16.STATEMENT

Parent Class: STANDARD OBJECT

Objects of class STATEMENT represent individual CIL statements. As with other objects, no specific implementation for statements is specified: they may be character strings or something more elaborate. This class is provided in the standard set chiefly to establish the maximum length of a statement. The maximum length of a statement imposes limits on the length of several other entities in the language, including object names, symbols, and so on.

3.2.16.1.Class Attributes

MAX LENGTH : INTEGER

The maximum number of characters allowed in a statement.

3.2.16.2.Instance Attributes

There are no instance attributes for class STATEMENT.

3.2.16.3.Operator Actions

There are no operator actions for class STATEMENT.

3.2.16.4.Command Actions

There are no commands for class STATEMENT.

3.2.17.PROCEDURE

Parent Class: STANDARD OBJECT

Objects of class PROCEDURE represent executable CIL procedures. See Section B-6 for more information on CIL procedures.

3.2.17.1.Class Attributes

There are no class attributes for class PROCEDURE.

3.2.17.2.Instance Attributes

There are no instance attributes for class PROCEDURE.

3.2.17.3.Operator Actions

There are no operator actions for class PROCEDURE.

3.2.17.4.Command Actions

PERFORM

The Perform command initiates execution of a procedure. If this command is issued from within a procedure, then execution of the caller is suspended and the called procedure is executed; when the called procedure completes execution, control is returned to the caller which resumes execution with the next statement in sequence.

A procedure may have zero, one or more qualifiers associated with it. The qualifiers that are allowed for a specific procedure are given in the declaration of formal parameters for the procedure.

Examples

Perform EVASIVE MANUEVER ALPHA
Perform PLANT GROWTH EXPERIMENT With SAMPLE TYPE:=RASPBERRY,

TERMINATE

This command terminates the procedure that is the object of the command. There are no qualifiers for this command.

Example

Termiinate PLANT GROWTH EXPERIMENT

SUSPEND

This command causes the procedure that is the object of the command to enter a wait state. Execution can only be restarted by a Resume command.

Example

Suspend PLANT GROWTH EXPERIMENT

RESUME

This command restarts the execution of a procedure that is in a wait state, either because of a Wait statement within the procedure or because of a Suspend command.

Example

Resume PLANT GROWTH EXPERIMENT